

Dipartimento di Scienze Aziendali - Management & Innovation Systems
Corso di Laurea in Data Science & Gestione dell'Innovazione

Il Gabibbo: una figura controversa

Come il Gabibbo ha creato un esercito di neofascisti nell'Italia del
21esimo secolo

INFORMATION DISORDER AND GLOBAL SECURITY

Echoes of the right: La genesi delle community altright su Telegram

Come i network digitali hanno forgiato l'altright su Telegram

by

Denise Brancaccio - MAT. 0222800163
Lucia Brando - MAT. 0222800162
Bruno Maria Di Maio - MAT. 0222800149

Instructor: G. Fenza
Teaching Assistant: D. Cavalieri
Project Duration: Month, Year – Month, Year
Faculty: Data Science & Gestione dell’Innovazione



Prefazione

Il mondo delle community online ha assunto un ruolo sempre più centrale nella formazione delle opinioni, nella condivisione di informazioni e nella costruzione di identità collettive. Con la crescente diffusione di notizie false e la polarizzazione delle opinioni, è cruciale comprendere le dinamiche di queste community e il loro rapporto con la qualità delle fonti informative.

Questo lavoro si propone di esplorare il livello di apertura delle community su Telegram e il legame tra tale apertura e l'attendibilità dei siti di notizie condivisi al loro interno.

L'obiettivo è non solo mappare le connessioni tra le community e le fonti, ma anche fornire uno strumento di analisi visiva che evidenzi le dinamiche di interazione attraverso l'utilizzo di grafi generati con Gephi.

Questo studio si inserisce in un contesto di ricerca multidisciplinare che combina data science, analisi delle reti sociali e valutazione delle fonti informative, offrendo nuove prospettive sulla comprensione dell'ecosistema informativo digitale.

Indice

1 Introduzione	4
1.1 Obiettivo principale	4
2 Metodi e strumenti di lavoro	5
2.1 Glossario	5
2.2 Telegram	7
2.3 NewsGuard	8
2.4 Gephi	8
2.5 Open Measures	9
3 Sviluppo del lavoro	10
3.1 Raccolta e pre-analisi dei dati	10
3.1.1 File newsguard.csv	10
3.1.2 File Open Measures.py	11
3.1.3 File main.py	12
3.1.4 File gephi.py	15
3.1.5 File node.csv:	16
3.1.6 File archs.csv:	17
3.1.7 File connections.py	17
3.1.8 File closure.py	19
3.1.9 File community.py	20
4 Grafi	25
4.1 Creazione dei grafi	25
4.1.1 Grafo orientato	25
4.1.2 Grafo non orientato	26
4.2 Analisi dei grafi	26
4.2.1 Analisi del grafo orientato	26
4.2.2 Analisi del grafo non orientato	31
4.3 Analisi dei risultati	31
5 Clustering	31
6 Conclusione e valutazione finale del team	31
6.1 Sintesi dei risultati	31

1 Introduzione

Negli ultimi anni, la diffusione di informazioni sui social media e sulle piattaforme di messaggistica istantanea ha radicalmente trasformato il modo in cui le persone accedono alle notizie.

Telegram, in particolare, è emerso come uno degli strumenti più utilizzati per la condivisione di contenuti informativi, grazie alla sua flessibilità e alla capacità di ospitare gruppi e canali con migliaia di membri. Tuttavia, questa libertà d'uso si accompagna a rischi significativi, tra cui la proliferazione di notizie false e l'amplificazione di narrazioni manipolatorie. Le **community online** rappresentano un microcosmo di dinamiche sociali e informative, dove l'apertura – intesa come la capacità di interagire con fonti e utenti esterni – gioca un ruolo cruciale nel determinare la qualità e l'affidabilità delle informazioni condivise.

Questo studio si propone di investigare il rapporto tra il livello di apertura delle community e l'attendibilità delle fonti giornalistiche che vi circolano, offrendo una panoramica approfondita delle interazioni tra utenti, contenuti e fonti informative.

1.1 Obiettivo principale

L'obiettivo principale di questo lavoro è analizzare il livello di apertura delle community presenti su Telegram e correlare tale apertura all'affidabilità dei siti di notizie condivisi.

Attraverso l'utilizzo di dati raccolti direttamente da **Telegram**, incrociati con valutazioni fornite da **NewsGuard**, si mira a costruire un grafo rappresentativo delle connessioni e delle interazioni tra community e fonti informative.

Il grafo, realizzato tramite **Gephi**, consente una visualizzazione chiara e intuitiva delle dinamiche emerse, facilitando l'identificazione di pattern significativi e relazioni critiche.

2 Metodi e strumenti di lavoro

Per condurre l'analisi, è stato sviluppato uno **script Python** progettato per raccogliere dati da Telegram.

Questo script ha permesso di filtrare i messaggi contenenti link a siti di notizie.

I dati raccolti sono stati elaborati confrontandoli con i punteggi di **NewsGuard**, generando così un file .csv che includeva nodi (le community) e archi (le connessioni tra essi).

Il file è stato poi importato in **Gephi** per creare un grafo che rappresentasse visivamente le dinamiche informative.

2.1 Glossario

Di seguito si riportano alcuni dei termini chiave utilizzati nel contesto della comunicazione e della diffusione delle informazioni.

Sono stati utilizzati termini che identificano contenuti di attualità e aggiornamenti di rilevante importanza:

- News
- Breaking
- Alert
- Headline
- Update
- Broadcast

Termini che si riferiscono a diverse tipologie di documenti informativi e giornalistici:

- Report
- Article
- Scoop
- Flash
- Bulletin
- Interview

Termini che indicano fenomeni informativi trasmessi in tempo reale o caratterizzati da una rapida diffusione:

- Live
- Trending
- Viral
- Insight
- Timeline
- Profile

- Community

Termini che descrivono situazioni di urgenza o emergenza:

- Emergency
- Incident
- Accident
- Disaster
- Crisis

Sono stati utilizzati, inoltre, temi di rilevanza sociale e politica:

- Conflict
- War
- Pandemic
- Health
- Technology
- Economy
- Politics
- Elections
- Debate
- Protest

Ma anche termini che delineano strumenti, processi e dinamiche legati alla raccolta, elaborazione e diffusione dell'informazione:

- Survey
- Poll
- Results
- Statistics
- Scandal
- Investigation
- Inquiry
- Revelation
- Discovery
- Press conference
- On Air

2.2 Telegram



Telegram è una piattaforma di messaggistica molto utilizzata per la creazione di gruppi e canali tematici in cui gli utenti possono condividere e discutere contenuti.

A differenza di altre piattaforme, Telegram permette la creazione di canali pubblici e gruppi con un numero potenzialmente illimitato di membri, favorendo una rapida diffusione delle informazioni.

L'architettura di Telegram, infatti, consente agli utenti di condividere link a siti web, creando un ecosistema in cui notizie, articoli e contenuti virali possono diffondersi rapidamente. Questo lo rende un campo di studio ideale per analizzare la diffusione delle notizie, soprattutto in relazione alla qualità delle informazioni condivise.

Inoltre, la disponibilità di API permette di raccogliere dati in modo automatizzato e strutturato, semplificando l'analisi dei messaggi e dei contenuti condivisi all'interno dei gruppi e canali.

Per il nostro studio, Telegram è stato scelto come piattaforma principale per monitorare e raccogliere i dati, in quanto fornisce un ampio spettro di community diverse e informazioni su un vasto numero di argomenti, spesso trattati in modo non filtrato e senza verifica.

2.3 NewsGuard



NewsGuard è uno strumento che fornisce valutazioni affidabili sui siti web di notizie, assegnando loro un punteggio in base a criteri come la trasparenza, la correttezza e l'imparzialità dei contenuti.

Ogni sito analizzato viene esaminato da un team di giornalisti professionisti, che valuta i parametri di qualità delle notizie pubblicate e fornisce una valutazione che aiuta gli utenti a discernere tra fonti affidabili e quelle che potrebbero diffondere disinformazione.

L'integrazione di NewsGuard nel nostro studio è stata essenziale, in quanto ha permesso di incrociare i link raccolti da Telegram con una valutazione professionale e oggettiva della loro affidabilità.

Grazie a questo strumento, è stato possibile determinare se le fonti di notizie condivise all'interno delle community fossero credibili o se provenissero da siti tendenti alla disinformazione. L'utilizzo di NewsGuard ha aggiunto un ulteriore livello di analisi alla nostra ricerca, consentendo di classificare i dati in modo oggettivo e basato su criteri di qualità giornalistica.

2.4 Gephi



Gephi è un software open-source utilizzato per l'analisi e la visualizzazione di grafi e reti complesse.

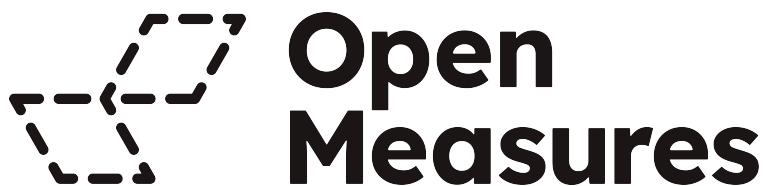
È particolarmente utile per studiare le relazioni tra entità attraverso la rappresentazione visiva di nodi e archi, dove ogni nodo rappresenta un'entità (in questo caso, una community), mentre gli archi rappresentano le connessioni tra di esse.

In questo studio, Gephi è stato utilizzato per visualizzare le connessioni tra i diversi nodi, permettendo una rappresentazione grafica che mostra la struttura e l'interconnessione di queste reti.

L'importazione dei dati raccolti e strutturati in formato CSV ha consentito di generare un grafo interattivo che rende facilmente visibili le relazioni tra le community e le fonti di notizie, offrendo al contempo una panoramica immediata delle dinamiche di diffusione delle informazioni.

Grazie alla sua capacità di gestire grandi quantità di dati e generare visualizzazioni dinamiche e interattive, Gephi ha svolto un ruolo cruciale nell'interpretazione e nella presentazione dei risultati, consentendo di osservare pattern, cluster e altre caratteristiche significative nelle connessioni tra le community e i siti di notizie.

2.5 Open Measures



Open Measures è una piattaforma avanzata progettata per supportare l'analisi e il monitoraggio delle interazioni sui social media, con un focus particolare sui contenuti generati dagli utenti.

La piattaforma offre API flessibili e potenti, che consentono di raccogliere dati strutturati su vasta scala.

Questi dati includono messaggi, metadati (ad esempio, autore, data e ora), contenuti multi-mediali, e informazioni sulle relazioni tra utenti o entità.

Le API di Open Measures sono progettate per essere altamente configurabili, permettendo di personalizzare le query attraverso parametri come parole chiave, intervalli temporali, e piattaforme social specifiche. Ciò consente agli utenti di accedere a dataset precisi e pertinenti per le loro analisi.

Ad esempio, è possibile eseguire ricerche avanzate per identificare discussioni che menzionano URL, frasi chiave o hashtag, oppure per analizzare contenuti pubblicati su piattaforme come Telegram, Twitter o Reddit.

Grazie alla sua architettura scalabile, Open Measures è particolarmente utile per progetti accademici, aziendali o governativi che richiedono l'analisi di grandi quantità di dati social.

Le sue applicazioni spaziano dal monitoraggio della disinformazione e dell'hate speech, all'analisi dei trend, fino agli studi sull'engagement e sulla polarizzazione delle opinioni.

Inoltre, la possibilità di esportare i dati in formati standard, come CSV, facilita l'integrazione con strumenti di analisi esterni come Python, R o software di visualizzazione come Gephi.

3 Sviluppo del lavoro

Il presente lavoro è articolato in diverse sezioni che riflettono il percorso di analisi intrapreso. Il seguente capitolo descriverà in dettaglio la metodologia adottata, illustrando le tecniche di raccolta, filtraggio e analisi dei dati.

Il successivo capitolo presenterà i risultati ottenuti, con particolare attenzione alla mappatura delle community e alla visualizzazione dei grafi.

Infine, verranno discussi i risultati evidenziando le implicazioni pratiche e teoriche, oltre a considerare i limiti dello studio e le potenziali direzioni future.

3.1 Raccolta e pre-analisi dei dati

La raccolta e la preparazione dei dati costituiscono la base fondamentale di questo studio, che mira ad analizzare il comportamento delle community su Telegram.

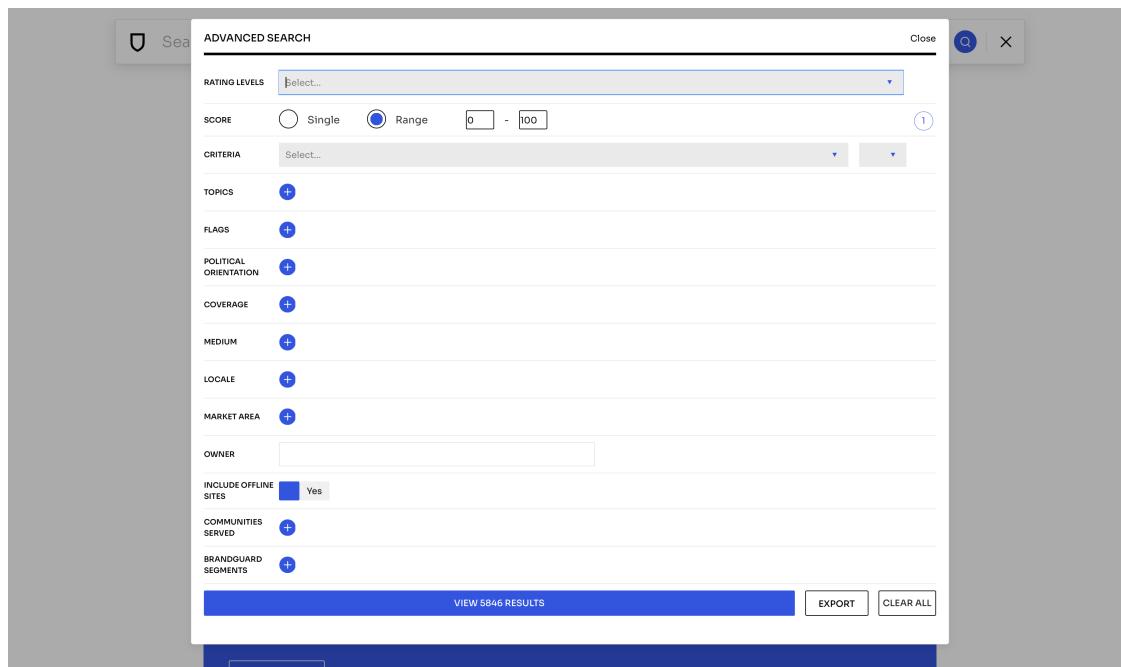
In questa fase, è stata utilizzata una combinazione di strumenti di programmazione e dataset per strutturare le informazioni raccolte, filtrare i dati rilevanti e prepararli per l'analisi dei grafi.

3.1.1 File newsguard.csv

Per l'avvio del lavoro è stato utilizzato il file "**newsguard.csv**", che contiene dati relativi a fonti di notizie.

L'unico criterio applicato per filtrare i dati in fase iniziale è stato rappresentato dagli score assegnati alle fonti, variabili su una scala da 0 a 100.

Questo approccio ha permesso di selezionare le fonti in base al loro punteggio senza ulteriori restrizioni.



3.1.2 File Open Measures.py

Per questo progetto è risultato fondamentale l'utilizzo dell'API di Open Measures. Grazie a quest'ultima, infatti, è stato possibile accedere a grandi quantità di dati strutturati relativi a Telegram, includendo metadati come contenuti dei messaggi, timestamp ed informazioni sugli autori.

Per configurare il sistema, è necessario:

- Ottener un token di autorizzazione (salvato nel file **open-measures-key.txt**).
- Definire i parametri di ricerca attraverso file JSON (**parametri/terms.json**), che specificano le colonne richieste, i tipi di dati e i termini di ricerca. Ogni termine viene verificato: se è già stato processato, non viene ricercato nuovamente.

Un esempio di chiamata API con la funzione **fetch_results** mostra la struttura del processo di raccolta:

```
1     def fetch_results(term, social, start_date, end_date,
2                         attempt_count=None):
3         if attempt_count is None:
4             attempt_count = {"count": 1}
5
6         current_attempt = attempt_count["count"]
7
8         params = {
9             'term': '(message:http OR message:https) AND message',
10            ':%s' % term,
11            'limit': 10000,
12            'site': social,
13            'since': start_date,
14            'until': end_date,
15            'esquery': 'true'
16        }
```

La funzione genera query dinamiche per l'API, suddivide automaticamente gli intervalli temporali in caso di limiti di risultati, e converte le risposte in un DataFrame pandas.

Quando il limite massimo di 10.000 risultati viene superato, l'intervallo temporale della query viene automaticamente suddiviso a metà.

Questo processo si ripete ricorsivamente fino a quando ogni intervallo produce meno di 10.000 risultati.

```
1     ...
2     if len(hits) == 10000:
3         mid_date = pd.Timestamp(start_date) + (pd.Timestamp(end_date)
4                                         - pd.Timestamp(start_date)) / 2
5
6         ...
7         first_half_df = fetch_results(term, social, start_date,
8                                         mid_date, attempt_count)
9         second_half_df = fetch_results(term, social, second_half,
10                                         end_date, attempt_count)
```

```

7     ...
8     else:
9         df = pd.DataFrame([hit['_source'] for hit in hits])
10    return df

```

Questa funzione suddivide gli intervalli temporali finché i dati non sono gestibili, garantendo una raccolta completa e robusta.

Infine, i risultati sono stati salvati in file CSV per ogni termine analizzato, come mostrato nel seguente frammento di codice:

```

1     output_file = "csv/social_termine_2024.csv"
2     df.to_csv(output_file, index=False)

```

Questa metodologia ha garantito la scalabilità del processo di raccolta e la creazione di una base dati robusta per le successive fasi di analisi.

I file finali generati sono in formato CSV e includono:

- **Colonne strutturate:** metadati come timestamp, contenuto dei messaggi, autori, ecc.
- **Dati consistenti:** grazie alla conversione automatica dei tipi di dati specificati nel file `dtypes.json`.

Un termine come "disinformazione", ricercato su Telegram per gennaio 2024, ha prodotto dati relativi a messaggi contenenti URL (message:http OR message:https) e salvati nel file CSV corrispondente. Grazie all'automazione, ogni intervallo temporale è stato gestito in modo autonomo, senza intervento manuale.

3.1.3 File main.py

Il sistema presentato analizza messaggi provenienti da Telegram per individuare e filtrare URL che puntano a domini di interesse, forniti da un file di riferimento (Newsguard).

Il processo include la lettura, il filtraggio e l'aggregazione dei dati in due output principali: **output.csv**, che raccoglie i messaggi rilevanti, e **occorrenze.csv**, che riassume statistiche di dominio.

Gli obiettivi del sistema sono:

- Identificare i domini di interesse nei messaggi di Telegram
- Analizzare quantitativamente il numero di occorrenze, visualizzazioni (views) e inoltri (forwards)
- Generare output strutturati e pronti per analisi successive

Gli URL vengono estratti dai messaggi tramite la funzione **extract_urls**, che normalizza i formati non standard (es. **[DOT]** → **.**) e identifica gli URL.

```

1     def extract_urls(text):
2         url_pattern = r"https?:\/\/(?:[-\w.]+(?:%[\da-fA-F]{2}){2})+"

```

```
3     if isinstance(text, str):
4         cleaned_message = re.sub(r"\[(DOT|dot|\.)\]", ".", str(text))
5         cleaned_message = re.sub(r"\](?=\\s|$)", "", cleaned_message)
6         return re.findall(url_pattern, cleaned_message)
7     return []
```

I domini vengono estratti dagli URL utilizzando la **libreria urllib.parse** per separare la parte rilevante.

```
1 def extract_domain(url):
2     try:
3         parsed_url = urlparse(url)
4         domain = parsed_url.netloc
5         if domain.startswith("www."):
6             domain = domain[4:]
7         return domain.lower()
8     except:
9         return ""
```

La funzione `process_csvs` analizza i file CSV nella directory `csv/` e li confronta con i domini di interesse presenti nel file `Newsguard`.

```
1 def process_csvs(output_filename, occorrenze_filename,
2     newsguard_filename="Newsguard.csv", folder_path="csv"):
3     newsguard_df = pd.read_csv(newsguard_filename).dropna(subset
4         =["Score"]).drop_duplicates()
5
6     original_domains = {domain.lower(): domain for domain in
7         newsguard_df["Domain"] if isinstance(domain, str)}
8     domain_scores = {domain.lower(): score for domain, score in
9         zip(newsguard_df["Domain"], newsguard_df["Score"]) if
10        isinstance(domain, str)}
11    domain_orientations = {domain.lower(): orientation for domain
12        , orientation in zip(newsguard_df["Domain"], newsguard_df[
13            "Orientation"]) if isinstance(domain, str)}
14
15    newsguard_domains = set(original_domains.keys())
16    columns_to_keep = ["message", "channelusername", "reactions",
17        "forwards", "postauthor", "replies", "views"]
18
19    matching_rows = []
20    domain_stats = defaultdict(lambda: {"occorrenze": 0, "views": 0,
21        "forwards": 0})
22
23    for filename in os.listdir(folder_path):
24        if filename.endswith(".csv"):
25            print("Elaborazione file: %s" % filename)
26            try:
```

```

18 df = pd.read_csv(os.path.join(folder_path, filename),
19                  low_memory=False).drop_duplicates()
20 for _, row in df.iterrows():
21     if isinstance(row["message"], str):
22         urls = extract_urls(row["message"])
23         domains = {extract_domain(url) for url in urls}
24         matching_domains = domains & newsguard_domains
25         if matching_domains:
26             matching_rows.append(row[columns_to_keep])
27             views = int(row["views"]) if pd.notna(row["views"]) else 0
28             forwards = int(row["forwards"]) if pd.notna(row["forwards"])
29             else 0
30             for domain in matching_domains:
31                 domain_stats[domain]["occorrenze"] += 1
32                 domain_stats[domain]["views"] += views
33                 domain_stats[domain]["forwards"] += forwards
34             except Exception as e:
35                 print("Errore nel processare %s: %s" % (filename, str(e)))
36
37             if matching_rows:
38                 output_df = pd.DataFrame(matching_rows)
39                 output_df.to_csv(output_filename, index=False)
40                 print("\nAnalisi completata. File %s creato con %s"
41                       " corrispondenze." % (output_filename, len(output_df)))
42             else:
43                 print("\nNessuna corrispondenza trovata.")
44
45             occorrenze_data = []
46             for domain_lower, stats in domain_stats.items():
47                 if stats["occorrenze"] > 0:
48                     occorrenze_data.append({
49                         "Domain": original_domains[domain_lower],
50                         "Score": domain_scores[domain_lower],
51                         "Orientation": domain_orientations[domain_lower],
52                         "Occhorrenze": stats["occorrenze"],
53                         "Views": stats["views"],
54                         "Forwards": stats["forwards"]
55                     })
56
57             if occorrenze_data:
58                 occorrenze_df = pd.DataFrame(occorrenze_data)
59                 occorrenze_df = occorrenze_df.sort_values("Occhorrenze",
60                     ascending=False)
61                 occorrenze_df.to_csv(occorrenze_filename, index=False)
62                 print("File %s creato con %s domini attivi.\n" %
63                     (occorrenze_filename, len(occorrenze_df)))
64             else:

```

```
print("Nessun dominio con occorrenze trovato.\n")
```

Le tipologie di output che si ottengono sono le seguenti:

- **output.csv:** Fornisce una visione dettagliata dei messaggi corrispondenti, utile per analisi qualitative e per approfondire il contesto dei domini rilevati.
- **occorrenze.csv:** Riassume i dati quantitativi, ordinando i domini per numero di occorrenze in ordine decrescente, evidenziando i domini più rilevanti.

Domain	Score	Orientation	Occorrenze	Views	Forwards
dominio1.com	60	Neutral	120	1500	300
dominio2.com	90	Left	75	800	120
dominio3.com	40	Right	45	500	100

3.1.4 File gephyp.py

Il file gephyp.py è uno script che processa i dati presenti nel file output.csv per generare due file output:

- **nodes.csv:** contiene l'elenco dei nodi ed il loro tipo
- **archs.csv:** contiene le relazioni tra i nodi

Questi file sono progettati per essere compatibili con Gephi e rappresentano rispettivamente i nodi (utenti e canali) e le relazioni (interazioni) tra di essi. Lo script ha lo scopo di:

- **Identificare i nodi:** creare un elenco di utenti e canali
- **Generare le relazioni:** costruire gli archi che rappresentano interazioni, come risposte tra utenti e canali
- **Salvare i risultati**

Lo script si articola in quattro sezioni principali.

La prima è la lettura del file **output.csv**:

```
1 # Lettura del file CSV
2 data = pd.read_csv("output.csv")
3 df = pd.DataFrame(data)
```

Per ogni riga del file, lo script analizza la **colonna replies**, che contiene le informazioni sugli utenti che hanno risposto a un determinato canale.

Se la colonna **replies** è un dizionario valido, vengono estratti gli identificatori univoci degli utenti (**user_id**) e i canali a cui hanno risposto.

Questi dati vengono salvati nel dizionario **dizionario**.

```

1      dizionario = {}
2
3      for index, replies in enumerate(df["replies"]):
4          # Converti la stringa in un dizionario
5          if isinstance(replies, str):
6              replies = ast.literal_eval(replies)
7              if isinstance(replies, dict) and replies["recent_repliers"]:
8                  for user, _ in enumerate(replies["recent_repliers"]):
9                      # Estrai l'ID dell'utente
10                     if "user_id" in replies["recent_repliers"][user]:
11                         user_id = replies["recent_repliers"][user]["user_id"]
12                         channel = df["channelusername"][index]
13                         # Aggiungi l'utente e il canale al dizionario
14                         if user_id not in dizionario:
15                             dizionario[user_id] = ("User", [channel])
16                         else:
17                             dizionario[user_id][1].append(channel)

```

I dati elaborati, come già detto in precedenza vengono salvati in due file csv.

3.1.5 File node.csv:

```

1      nodes = "nodes.csv"
2      target_set = set()
3
4      with open(nodes, mode="w", newline="", encoding="utf-8") as
5          file:
6              writer = csv.writer(file)
7              writer.writerow(["id", "type"]) # Intestazione
8
9              for key, (value, target) in dizionario.items():
10                  writer.writerow([key, value]) # Nodo utente
11                  for element in target:
12                      if element not in target_set:
13                          target_set.add(element)
14                          writer.writerow([element, "Channel"]) # Nodo canale

```

ID	Type
12345678	User
channel_1	Channel

3.1.6 File archs.csv:

```
1     archs = "archs.csv"
2
3     with open(archs, mode="w", newline="", encoding="utf-8") as
4         file:
5             writer = csv.writer(file)
6             writer.writerow(["source", "target", "type"]) # Intestazione
7
8             for key, (_, target) in dizionario.items():
9                 for element in target:
10                     writer.writerow([key, element, "Reply"]) # Arco tra utente e
11                         canale
```

Source	Target	Type
12345678	channel_1	Reply

3.1.7 File connections.py

Il file **connections.py** è una componente essenziale del progetto che estende le relazioni presenti nel dataset **archs.csv**.

Il suo scopo è generare nuove connessioni tra i nodi basandosi su combinazioni esistenti e consolidare tali relazioni in un uovo file csv denominato **archs_new.csv**.

Lo script esegue le seguenti operazioni:

- **Caricamento del dataset originale:** utilizza il file archs.csv come input per leggere le connessioni esistenti
- **Raggruppamento per origine:** per ogni nodo sorgente, individua i nodi destinazione collegati (source->target)
- **Generazione di combinazioni:** crea tutte le possibili coppie tra i nodi di destinazione collegati a una stessa sorgente
- **Consolidamento delle connessioni:** concatena le connessioni generate con quelle originali e salva il risultato in **archs_new.csv**

La prima parte del codice legge il dataset per manipolare i dati tabulari:

```
1     import pandas as pd
2     from itertools import combinations
3
4     # Leggi il file
5     df = pd.read_csv("archs.csv")
```

Il file **archs.csv** contiene le connessioni esistenti, strutturate con colonne **source**, **target** e **type**, dove:

- **Source:** rappresenta il nodo sorgente
- **Target:** rappresenta il nodo destinazione
- **Type:** specifica la natura della relazione

Successivamente, si va ad inizializzare le strutture per le nuove connessioni:

```

1      # Nome del file di output
2      output_file = "archs_new.csv"
3
4      # Inizializza un DataFrame vuoto
5      df_total_channels = pd.DataFrame(columns=["source", "target",
6                                              "type"])

```

Lo script raggruppa i dati per source, estraendo per ciascun nodo sorgente l'elenco dei nodi di destinazioni associati.

Successivamente, utilizza la **funzione combinations** della libreria **itertools** per generare tutte le coppie possibili tra questi nodi di destinazione:

```

1      for _, group in df.groupby("source"):
2          # Crea un set di utenti per evitare duplicati
3          targets = set(group["target"])
4
5          # Genera le combinazioni
6          pairs = list(combinations(targets, 2))
7          df_new = pd.DataFrame(pairs, columns=["source", "target"]).
8              drop_duplicates()
9
10         # Aggiungi la colonna "type" con valore "Connection"
11         df_new["type"] = "Connection"
12
13         # Concatena al DataFrame totale
14         df_total_channels = pd.concat([df_total_channels, df_new],
15                                         ignore_index=True)

```

In questo processo i nodi di destinazione vengono memorizzati in un set (**targets**) per garantire l'unicità e le combinazioni di coppie vengono generate per creare nuove connessioni indirette tra i nodi.

Le connessioni aggiuntive vengono unite a quelle originali presenti nel dataset:

```

1      # Concatena il DataFrame totale al DataFrame originale
2      df_final = pd.concat([df, df_total_channels], ignore_index=
3                             True).dropna()
4      df_final.to_csv(output_file, index=False)

```

Il risultato è salvato in un nuovo file **archs_new.csv** che contiene sia le connessioni originali sia quelle derivate.

Source	Target	Type
user123	canale_1	Reply
user123	canale_2	Reply
user456	canale_3	Reply

Tabella 1: Esempio di file ‘archs.csv’.

Source	Target	Type
user123	canale_1	Reply
user123	canale_2	Reply
user456	canale_3	Reply
canale_1	canale_2	Connection

Tabella 2: Esempio di file ‘archs_new.csv’ con connessioni derivate.

3.1.8 File closure.py

In questa sezione verrà analizzato e descritto nel dettaglio il codice sviluppato per calcolare un indice denominato *"tasso di chiusura"* per diverse community, andandosi a basare sulle interazioni tra utenti. Iniziamo con il caricare i dati:

```

1 df = pd.read_csv("archs_new_2.csv")
2 df = df[df.type == "Reply"]
3 df_community = pd.read_csv("community.csv")

```

- **archs_new_2.csv:** contiene informazioni sulle interazioni tra utenti, inclusi i tipi di messaggi inviati. Solo i messaggi di tipo "Reply" vengono considerati per le successive analisi.
- **community.csv:** fornisce dettagli sulle community, come il numero di messaggi e attributi associati.

I file csv vengono caricati in due dataframe denominati **df** e **df_community**.

Si prosegue, poi, con il calcolo degli utenti per canale:

```

1 users_per_channel = df.groupby("target")["source"].apply(set)
      .to_dict()

```

Questo passaggio raggruppa i dati in base ai canali destinatari (**target**) e crea un dizionario in cui ogni chiave rappresenta un canale, mentre il valore è l’insieme degli utenti (**source**) che hanno interagito con quel canale.

Si passa al calcolo dei canali per utente:

```

1 channels_per_user = df.groupby("source")["target"].apply(set)
      .to_dict()

```

In modo simile, questo codice calcola per ogni utente l’insieme dei canali (**target**) con cui ha interagito. Il risultato è un dizionario in cui la chiave è un utente e il valore è un insieme

di canali.

Si calcola, poi, il tasso di chiusura:

```
1 closure_scores = {}
2     for channel, users in users_per_channel.items():
3         total_channels = sum(len(channels_per_user[user]) for user in
4             users)
5         closure_scores[channel] = len(users) / total_channels
```

Quest'ultimo si tratta di una metrica che quantifica la concentrazione degli utenti in un canale rispetto al numero totale di canali con cui interagiscono. Per ogni canale:

```
1 closure_scores = {}
2     for channel, users in users_per_channel.items():
3         total_channels = sum(len(channels_per_user[user]) for user in
4             users)
5         closure_scores[channel] = len(users) / total_channels
```

- Si somma il numero di canali associati a ciascun utente del canale corrente
- Il tasso di chiusura è definito come il rapporto tra il numero di utenti unici del canale e il numero totale di canali con cui questi utenti interagiscono

Ultimo passo è l'integrazione del tasso di chiusura precedentemente calcolato:

```
1 df_community["Tasso_di_chiusura"] = df_community["Community"]
2     .map(closure_scores)
```

I valori calcolati vengono mappati al DataFrame **df_community**, aggiungendo una nuova colonna denominata "Tasso di chiusura".

Solo le community che hanno almeno 200 messaggi vengono considerate:

```
1 df_community = df_community[df_community["Messaggi"] >= 200].
2     dropna(subset=["Tasso_di_chiusura"])
```

I dati filtrati e arricchiti con il tasso di chiusura vengono salvati in un nuovo file csv denominato **community_new.csv**.

3.1.9 File **community.py**

In questo paragrafo finale, si va ad analizzare in dettaglio il codice sviluppato per l'elaborazione e l'analisi dei dati delle community, esaminando ogni componente e la sua funzione specifica nel processo complessivo.

Il codice inizia con l'importazione delle librerie necessarie:

- **pandas**: per la manipolazione dei dataframe.
- **re**: per le operazioni con espressioni regolari.
- **defaultdict**: per la gestione di dizionari con valori predefiniti.

- **urlparse**: per l'analisi degli URL.
- **numpy**: per operazioni numeriche efficienti

Il codice definisce anche alcuni pattern regex ottimizzati per il parsing degli URL e la pulizia del testo:

```

1     URL_PATTERN = re.compile(r"https://(?:[-\w.]+(?:%[\da-fA-F][\w.])*)+")
2     DOT_PATTERN = re.compile(r"\[(DOT|dot|\.)\]")
3     BRACKET_PATTERN = re.compile(r"\](?=\\s|$)")

```

La funzione **preprocess_newsguard_data** si occupa di preparare i dati di Newsguard per l'analisi successiva:

```

1     def preprocess_newsguard_data(newsguard_df):
2         clean_df = newsguard_df.dropna(subset=["Score"]).
3             drop_duplicates()
4
5         return {
6             'original_domains': {d.lower(): d for d in clean_df["Domain"]
7                 if isinstance(d, str)},
8             'orientations': {d.lower(): o for d, o in zip(clean_df["Domain"], clean_df["Orientation"])
9                 if isinstance(d, str)},
10            'scores': {d.lower(): s for d, s in zip(clean_df["Domain"],
11                clean_df["Score"])
12                if isinstance(d, str)},
13            'topics': {d.lower(): t for d, t in zip(clean_df["Domain"],
14                clean_df["Topics"])
15                if isinstance(d, str)}
16        }
17    }

```

Questa funzione:

- Rimuove le righe con score mancanti e duplicati
- Crea quattro dizionari normalizzati per domini, orientamenti, score e topic
- Converte tutti i domini in minuscolo per garantire confronti case-insensitive
- Filtra eventuali valori non stringa per evitare errori

Le funzioni **extract_urls** e **extract_domain** lavorano in tandem per processare gli URL nei messaggi:

```

1     def extract_urls(text):
2         if not isinstance(text, str):
3             return []
4         cleaned = DOT_PATTERN.sub(".", text)
5         cleaned = BRACKET_PATTERN.sub("", cleaned)

```

```

6     return URL_PATTERN.findall(cleaned)
7
8     def extract_domain(url):
9         try:
10            parsed = urlparse(url)
11            domain = parsed.netloc
12            return domain[4:].lower() if domain.startswith("www.") else
13                domain.lower()
14            except:
15                return ""

```

Queste funzioni hanno gli scopi di:

- Normalizzano il testo sostituendo varianti di "dot" con punti
- Estraggono tutti gli URL validi dal testo
- Parsano gli URL per estrarre domini
- Rimuovono il prefisso "www." e convertono in minuscolo

Il vero cuore dell'analisi è la funzione **process_community_stats**:

```

1     def process_community_stats(df, newsguard_data):
2         community_stats = defaultdict(lambda: {"messaggi": 0, "views":
3             0, "forwards": 0, "domains": {}})
4         newsguard_domains = set(newsguard_data['original_domains'].keys())
5
6         df['views'] = pd.to_numeric(df['views'], errors='coerce').
7             fillna(0).astype(np.int32)
8         df['forwards'] = pd.to_numeric(df['forwards'], errors='coerce').
9             fillna(0).astype(np.int32)
10
11        for community, group in df.groupby('channelusername'):
12            valid_messages = group[group['message'].notna()]
13
14            for _, row in valid_messages.iterrows():
15                urls = extract_urls(row['message'])
16                if not urls:
17                    continue
18
19                domains = {extract_domain(url) for url in urls}
20                matching_domains = domains & newsguard_domains
21
22                if matching_domains:
23                    stats = community_stats[community]
24                    stats["messaggi"] += 1
25                    stats["views"] += row['views']
26                    stats["forwards"] += row['forwards']

```

```

24
25     for domain in matching_domains:
26         if domain in community_stats[community]["domains"]:
27             community_stats[community]["domains"][domain] += 1
28         else:
29             community_stats[community]["domains"][domain] = 1
30
31     return community_stats

```

Questa funzione:

- Inizializza un dizionario per memorizzare le statistiche delle community
- Converte e pulisce i dati numerici (views e forwards)
- Raggruppa i dati per community
- Analizza ogni messaggio per estrarre e contare i domini
- Aggiorna le statistiche per ogni dominio trovato che corrisponde ai dati Newsguard

Le funzioni `calculate_topic_percentages` e `calculate_orientation_percentages` elaborano le distribuzioni dei topic e degli orientamenti:

```

1   def calculate_topic_percentages(domains, topics_data):
2       all_topics = []
3       for domain in domains:
4           topic_str = topics_data.get(domain, '')
5           if isinstance(topic_str, str):
6               topics = [t.strip() for t in topic_str.split(',') if t.strip()]
7               all_topics.extend(topics)
8
9       if not all_topics:
10          return {}
11
12      topic_counts = {}
13      total_topics = len(all_topics)
14      for topic in set(all_topics):
15          count = all_topics.count(topic)
16          topic_counts[topic] = (count / total_topics) * 100
17
18      return dict(sorted(topic_counts.items(), key=lambda x: x[1],
19                        reverse=True))

```

Le due funzioni vanno a raccogliere tutti i topic/orientamenti dai domini rilevanti, calcolano le percentuali di occorrenza ed ordinano i risultati in ordine decrescente.

La funzione `calculate_community_data` aggrega tutti i dati elaborati:

```

1   def calculate_community_data(community_stats, newsguard_data):
2       :

```

```

2     community_data = []
3
4     for community, stats in community_stats.items():
5         if not stats['domains']:
6             continue
7
8         domains = stats['domains']
9
10        topic_percentages = calculate_topic_percentages(domains,
11                  newsguard_data['topics'])
12        orientation_percentages = calculate_orientation_percentages(
13                  domains, newsguard_data['orientations'])
14
15        valid_scores = [newsguard_data['scores'].get(domain)
16                      for domain in domains]
17        if domain in newsguard_data['scores']:
18            avg_score = sum(valid_scores) / len(valid_scores) if
19            valid_scores else 0
20
21        community_data.append({
22            "Community": community,
23            "Messaggi": stats["messaggi"],
24            "Views": stats["views"],
25            "Forwards": stats["forwards"],
26            "Score\u2022medio": avg_score,
27            "Top\u2022topic": max(topic_percentages.items(), key=lambda x:
28                x[1])[0] if topic_percentages else '',
29            "Top\u2022domain": max(domains.items(), key=lambda x: x[1])
30                [0],
31            "Orientamento\u2022percentuale": orientation_percentages,
32            "Topic\u2022percentuale": topic_percentages,
33            "Domini": domains,
34            "Frequent\u2022domains": dict(sorted(domains.items(),
35                key=lambda x: x[1],
36                reverse=True)[:5])
37        })
38
39
40    return community_data

```

Questa funzione va ad elaborare i dati per ogni community, calcola statistiche aggregate come score medio e top topic; infine, organizza i dati in un formato strutturato per l'output finale.

4 Grafi

4.1 Creazione dei grafi

Inizialmente, l'analisi era stata concepita per essere effettuata su grafi orientati, con l'obiettivo di sfruttare la direzionalità degli archi per rappresentare relazioni.

Tuttavia, durante lo sviluppo del lavoro, si sono presentate alcune limitazioni tecniche e metodologiche che hanno reso necessario spostare l'attenzione verso l'utilizzo di grafi non orientati, privilegiando una rappresentazione più semplice e simmetrica delle relazioni.

Per garantire una documentazione completa e trasparente, abbiamo comunque deciso di includere entrambe le versioni dell'analisi, offrendo un confronto tra i due approcci.

4.1.1 Grafo orientato

Il grafo riportato rappresenta una rete di utenti e comunità presenti su Telegram, modellata attraverso una struttura complessa di nodi e archi.

Questa rappresentazione è stata ottenuta utilizzando il software Gephi, attraverso il caricamento dei dataset **node.csv** e **archs.csv**, contenenti rispettivamente le informazioni relative ai nodi (utenti o comunità) e agli archi (connessioni o interazioni tra i nodi).

I nodi, visualizzati con colori distinti, identificano cluster di utenti che condividono connessioni più dense, suggerendo l'esistenza di comunità tematiche o gruppi coesi.

Gli archi, che collegano i nodi, rappresentano le interazioni o le connessioni dirette tra i diversi utenti o gruppi, evidenziando la struttura globale della rete.

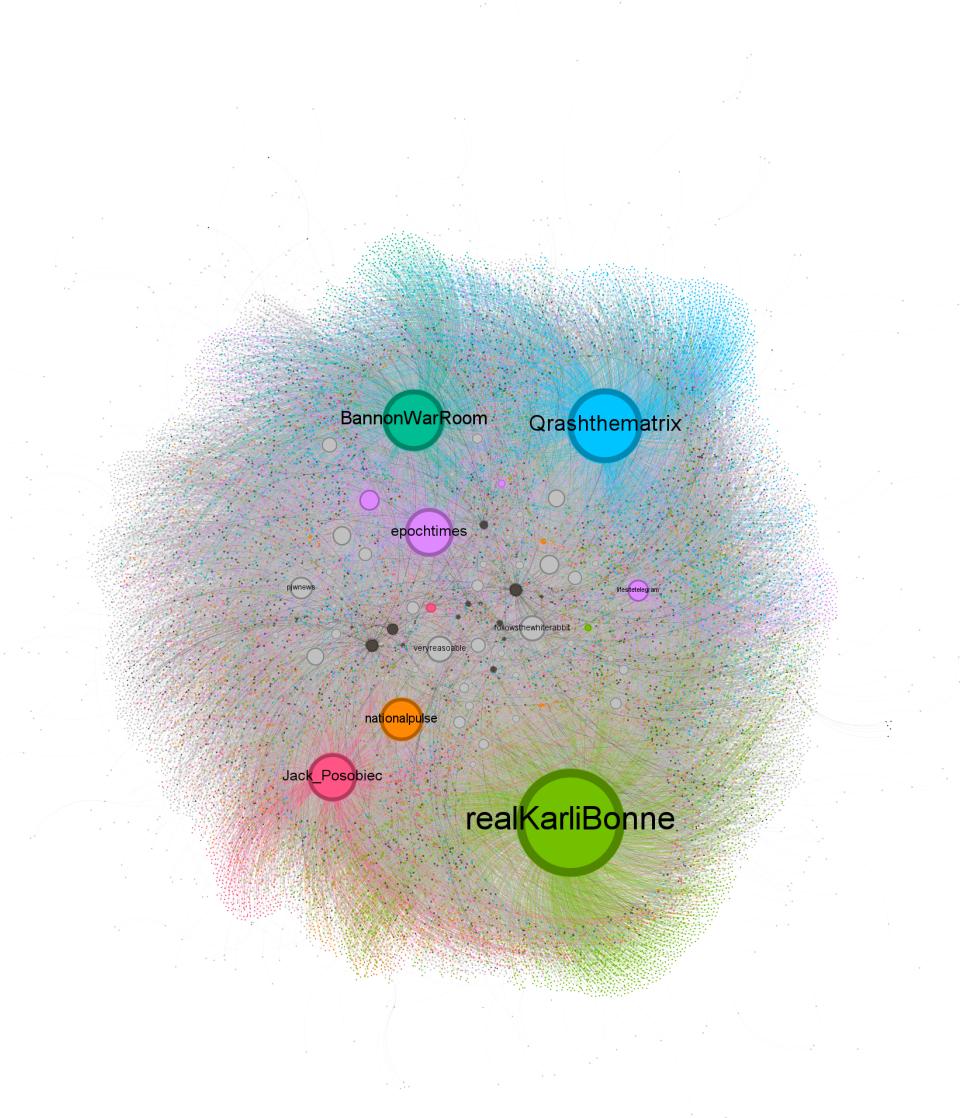
La distribuzione spaziale del grafo evidenzia una centralità maggiore in alcune aree, dove i nodi risultano più grandi e prominenti.

Questi possono essere interpretati come utenti o gruppi particolarmente influenti all'interno della rete, il cui ruolo potrebbe essere cruciale nella diffusione di informazioni o nel consolidamento di opinioni.

Al contrario, i nodi periferici, distanziati dal nucleo centrale, possono rappresentare utenti marginali o comunità meno integrate, contribuendo comunque alla diversità della rete complessiva.

La colorazione differenziale dei cluster permette una chiara identificazione delle comunità, facilitando l'analisi di fenomeni come la polarizzazione, l'influenza e la diffusione di informazioni all'interno del network.

L'osservazione visiva del grafo mostra come le comunità tendano a formare strutture relativamente compatte, con alcune interconnessioni tra gruppi distinti, suggerendo possibili punti di scambio informativo o di sovrapposizione tematica.



4.1.2 Grafo non orientato

4.2 Analisi dei grafi

4.2.1 Analisi del grafo orientato

Il seguente codice implementa un'analisi strutturale sul grafo orientato e esportato in formato **.graphml**.

Esso utilizza librerie Python come **NetworkX** per calcoli di metriche topologiche avanzate e **Pandas** per l'organizzazione e l'esportazione dei risultati.

Il grafo viene analizzato attraverso misure di centralità e metriche di rete.

Di seguito è riportata una spiegazione dettagliata:

○ Caricamento del grafo:

Il grafo viene caricato utilizzando il metodo **nx.read_graphml**, che importa dati

strutturati in formato **.graphml**. Una volta caricato, vengono stampate informazioni generali sulla rete, come il numero di nodi e di archi:

```
1     print("Grafo caricato con successo. Numero di nodi:",  
      graph.number_of_nodes(), "Numero di archi:", graph.  
      number_of_edges())
```

Questo consente una verifica preliminare per assicurarsi che i dati siano stati importati correttamente.

- **Filtraggio dei nodi "channel":** Il codice filtra i nodi con un **attributo "type"** pari a **"Channel"**, utilizzando una lista comprensione:

```
1     channels = [node for node in graph.nodes if graph.nodes[  
      node].get("type") == "Channel"]
```

Questa operazione seleziona un sottoinsieme specifico del grafo, così da poter focalizzare l'analisi su canali Telegram piuttosto che su altri tipi di nodi.

Il numero totale di nodi selezionati viene stampato per trasparenza.

- **Calcolo delle metriche di centralità:**

- **Degree:**

Il grado rappresenta il numero di connessioni di un nodo ed è calcolato senza considerare eventuali pesi sugli archi:

```
1     degree_dict = dict(graph.degree(channels))
```

Questa metrica è utile per identificare nodi altamente connessi all'interno della rete.

- **Betweenness Centrality:**

Utilizzando l'algoritmo ottimizzato di Brandes, il codice misura quanto un nodo si trovi su percorsi più brevi tra coppie di altri nodi.

Questo indica l'importanza del nodo come intermediario:

```
1     betweenness_dict = nx.betweenness_centrality(  
      graph, normalized=True, endpoints=False)
```

L'opzione **normalized=True** assicura che i valori siano scalati rispetto alla dimensione del grafo.

- **Closeness Centrality:**

Questa metrica misura quanto un nodo sia vicino a tutti gli altri nodi della rete, in termini di distanza geodetica:

```
1     closeness_dict = nx.closeness_centrality(graph)
```

- **Authority Score:**

Se il grafo è orientato, viene calcolata l'authority di ciascun nodo utilizzando

l'algoritmo HITS, che assegna punteggi in base alla qualità delle connessioni in entrata:

```
1     _, authority_dict = nx.hits(graph, normalized=True)
```

Nel caso di grafi non orientati, l'authority viene impostata a zero, poiché Gephi non supporta il calcolo di questa metrica per grafi non diretti.

- **Eigenvector Centrality:**

Questa metrica valuta l'influenza di un nodo in base all'importanza dei suoi vicini, calcolata iterativamente:

```
1     eigenvector_dict = nx.eigenvector_centrality(
2         graph, max_iter=1000, tol=1e-6)
```

Parametri come **max_iter** e **tol** sono scelti per garantire la convergenza.

○ Creazione dei risultati:

I risultati delle metriche vengono raccolti in una lista di dizionari, dove ciascun dizionario rappresenta un nodo e le relative metriche calcolate:

```
1     results = []
2     for node in channels:
3         results.append({
4             "Node": node,
5             "Degree": degree_dict.get(node, 0),
6             "Betweenness": betweenness_dict.get(node, 0),
7             "Closeness": closeness_dict.get(node, 0),
8             "Authority": authority_dict.get(node, 0),
9             "Eigenvector": eigenvector_dict.get(node, 0),
10            })
```

○ Esportazione risultati:

I risultati vengono convertiti in un DataFrame di Pandas, una struttura dati tabellare, e salvati in **formato .csv**:

```
1     df = pd.DataFrame(results)
2     output_file = "channel_metrics_gephi.csv"
3     df.to_csv(output_file, index=False)
```

La fase successiva dell'analisi del grafi prevede che i dati derivati vengano elaborati ulteriormente per calcolare una nuova metrica chiamata "**tasso di apertura**" (**Openness**).

Questo processo consente di arricchire ulteriormente le informazioni estratte, aggiungendo una dimensione sintetica che combina diverse metriche di centralità per fornire una visione complessiva dell'importanza e della posizione strategica dei nodi della rete.

Lo script si basa sui due file: il primo, **statpython.csv**, contiene i dati di base del grafo già processati e arricchiti con metriche di centralità; il secondo, **statpython_with_openness.csv**, viene generato dallo script ed include la nuova colonna "Openness" accanto alle metriche

originali, ampliando così le possibilità di analisi successive.

Il file **statpython.csv** rappresenta una tabella con informazioni dettagliate per ogni nodo del grafo. Le colonne principali includono:

- **Node:** identificativo univoco del nodo nella rete. Questo attributo consente di distinguere ciascun elemento analizzato, garantendo l'integrità dei collegamenti tra i dati originali e quelli derivati.
- **Degree:** numero di connessioni dirette che un nodo ha nella rete, una misura semplice ma fondamentale della sua interconnessione.
- **Betweenness:** misura dell'importanza di un nodo come intermediario nei percorsi brevi tra altri nodi, evidenziando il suo ruolo nel facilitare il flusso di informazioni.
- **Closeness:** rappresenta la vicinanza di un nodo a tutti gli altri nodi della rete, calcolata come l'inverso della somma delle distanze, una metrica utile per identificare nodi centrali.
- **Authority:** valore che indica l'autorevolezza del nodo basato sull'algoritmo HITS, particolarmente rilevante in grafi diretti dove l'influenza reciproca tra nodi gioca un ruolo cruciale.
- **Eigenvector:** centralità che misura l'influenza del nodo in base all'importanza dei nodi a cui è connesso, una metrica iterativa che considera il contesto globale della rete

Un esempio delle prime cinque righe di questo file è il seguente:

```
1     Node , Degree , Betweenness , Closeness , Authority , Eigenvector
2     1 , 10 , 0.02 , 0.4 , 0.3 , 0.8
3     2 , 5 , 0.01 , 0.35 , 0.2 , 0.7
4     3 , 8 , 0.03 , 0.38 , 0.25 , 0.75
5     4 , 6 , 0.015 , 0.37 , 0.22 , 0.72
6     5 , 7 , 0.02 , 0.36 , 0.24 , 0.73
```

Questa struttura garantisce la chiarezza e la coerenza necessarie per operazioni successive di analisi e calcolo.

Lo script inizia caricando i dati da **statpython.csv** utilizzando la **libreria pandas**, un potente strumento per la manipolazione di dati strutturati.

Dopo aver caricato i dati, verifica che le colonne necessarie siano presenti, garantendo che il dataset sia completo e conforme ai requisiti dell'analisi:

```
1     file_path = 'statpython.csv'
2     df = pd.read_csv(file_path)
3     required_columns = ["Node", "Degree", "Betweenness", "
4         Closeness", "Authority", "Eigenvector"]
5     if not all(col in df.columns for col in required_columns):
6         raise ValueError(f"Il file CSV deve contenere le seguenti
7             colonne: {', '.join(required_columns)}")
```

Se una delle colonne mancasse, lo script interromperebbe l'esecuzione con un messaggio di errore chiaro, assicurando un controllo di qualità sui dati in ingresso.

Per rendere comparabili le diverse metriche, lo script utilizza il metodo **MinMaxScaler** di **sklearn** per scalare i valori nel range [0,1].

Questa normalizzazione è essenziale per eliminare eventuali disparità dovute a scale diverse nelle metriche originali e garantire che ogni metrica contribuisca equamente al calcolo successivo:

```
1     metrics = ["Degree", "Betweenness", "Closeness", "Authority",
2                  "Eigenvector"]
3     scaler = MinMaxScaler()
4     normalized_metrics = scaler.fit_transform(df[metrics])
```

Il risultato è una matrice in cui ogni valore rappresenta una versione scalata del corrispondente valore originale, uniformando l'importanza relativa di ciascuna metrica.

La metrica "**Openness**" viene calcolata come la media aritmetica delle metriche normalizzate per ogni nodo.

Questo approccio aggrega informazioni complesse in un unico valore, semplificando l'identificazione di nodi che mostrano una combinazione equilibrata di centralità e influenza:

```
1 df["Openness"] = normalized_metrics.mean(axis=1)
```

Il risultato è una nuova colonna aggiunta al dataset, che fornisce un'indicazione sintetica ma potente del ruolo globale del nodo nella rete analizzata.

Il file aggiornato, contenente la nuova colonna "Openness", viene salvato in un nuovo file CSV.

Questa operazione consente di conservare i dati elaborati in un formato facilmente condivisibile e compatibile con ulteriori analisi:

```
1 output_file_path = "statpython_with_openness.csv"
2 df.to_csv(output_file_path, index=False)
```

Le prime cinque righe del file generato appaiono così:

```
1 Node, Degree, Betweenness, Closeness, Authority, Eigenvector,
2   Openness
3   1, 10, 0.02, 0.4, 0.3, 0.8, 0.508
4   2, 5, 0.01, 0.35, 0.2, 0.7, 0.432
5   3, 8, 0.03, 0.38, 0.25, 0.75, 0.488
6   4, 6, 0.015, 0.37, 0.22, 0.72, 0.465
7   5, 7, 0.02, 0.36, 0.24, 0.73, 0.482
```

Questa tabella arricchita permette di ottenere informazioni preziose in modo rapido ed efficiente.

Questo approccio permette di integrare informazioni complesse in un formato più facilmente interpretabile, utile per identificare nodi particolarmente influenti o strategici nella rete.

Inoltre, la combinazione dei due file consente una pipeline analitica ben definita, in cui ogni fase aggiunge un livello di approfondimento, migliorando la comprensione della struttura e delle dinamiche della rete.

4.2.2 Analisi del grafo non orientato

4.3 Analisi dei risultati

5 Clustering

6 Conclusione e valutazione finale del team

6.1 Sintesi dei risultati