



UNIVERSITA' DEGLI STUDI DI SALERNO

DIPARTIMENTO DI SCIENZE AZIENDALI - MANAGEMENT &
INNOVATION SYSTEMS

CORSO DI LAUREA MAGISTRALE IN DATA SCIENCE & GESTIONE
DELL'INNOVAZIONE

Project Work PxDS

A.A. 2024-2025
Docente: F. Orciuoli

Gruppo N.1
Candidati:

DENISE BRANCACCIO - MAT. 0222800163
LUCIA BRANDO - MAT.0222800162
BRUNO MARIA DI MAIO - MAT.0222800149

Indice

1	Introduzione	2
1.1	Project Work	2
1.2	Definizioni utilizzate	2
1.3	Organizzazione del gruppo	4
1.4	Tecnologie utilizzate	4
2	Descrizione del codice	5
2.1	Modelli	6
2.1.1	Sentiment Analysis	7
2.1.2	Emotion Analysis	7
2.1.3	Propaganda Classifier	7
2.1.4	Toxicity Classifier	7
2.1.5	Word Embedding	7
3	Sequence Diagram	8
3.1	Funzione def speech_info(politician,speech)	8
3.2	Funzione def detect_propaganda_type(text, classifier=propaganda_classifier, chunk_size=512, overlap=128)	8
3.3	Funzione def offsets(speech,data)	10
4	Funzioni	12
4.1	Verify Politician	12
4.2	Get Politician Info	12
4.3	Speech Info	15
4.4	Speech Data	15
4.5	Classify Text	16
4.6	Split Sentences	16
4.7	Propaganda Criteria	16
4.8	Detect Propaganda Type	17
4.9	Contains Toxicity Criteria	18
4.10	Classify Toxicity	19
4.11	Offsets	19
4.12	Formality	20
4.13	Readablity	20
4.14	Analyze Narrative Structure	21
4.15	TF-IDF	21
4.16	Word Embeddings	22
4.17	Visualize Embeddings	23
4.18	Analyze Persuasion	24
5	Analisi delle Prestazioni	27
6	Conclusioni	28
7	Collegamenti esterni	28

1 Introduzione

Questa documentazione descrive il lavoro svolto per l'esame di Programming for Data Science sviluppato dal nostro gruppo di lavoro, volto all'analisi automatizzata di discorsi politici. Dopo una panoramica della traccia assegnata, vengono presentate le tecnologie utilizzate, l'organizzazione del lavoro e le ipotesi applicative del sistema.

In seguito, nei successivi capitoli, vengono illustrate le funzionalità del codice, i diagrammi di sequenza per la soluzione dei requisiti, gli esempi pratici di utilizzo e l'analisi delle prestazioni. Infine, nell'ultima sezione, vengono riportati i risultati raggiunti e i link di riferimento per la repository Github del progetto.

1.1 Project Work

La traccia del progetto ha richiesto lo sviluppo di una data pipeline per l'arricchimento di un dataset chiamato speech-a.tsv, composto da tre tipi di contenuto: nomi di politici, i loro discorsi e un valore binario ad essi associato.

L'obiettivo finale è creare un nuovo dataset che supporti l'analisi e la detection di forme di propaganda all'interno dei discorsi. L'arricchimento prevede l'integrazione di informazioni sugli autori e i loro discorsi, l'aggiunta di feature linguistiche e contenutistiche come leggibilità, sentiment, parole chiave, e narrative sottostanti, oltre alla possibilità di individuare offset e tecniche specifiche di propaganda.

1.2 Definizioni utilizzate

In una prima fase, il team si è occupato dello studio e della definizione di due concetti fondamentali per lo svolgimento del lavoro: 'propaganda' e 'storytelling'. Per comprendere appieno cosa sia la propaganda, ci siamo riferiti alla seguente definizione fornita dal Dizionario di Politica di N. Bobbio, N. Matteucci e G. Pasquino (Torino, Utet Libreria, 2004, p. 775):

"La propaganda può essere definita come la diffusione deliberata e sistematica di messaggi indirizzati a un determinato uditorio, con l'obiettivo di creare un'immagine positiva o negativa di determinati fenomeni (persone, movimenti, eventi, istituzioni, ecc.) e di stimolare determinati comportamenti. La propaganda è quindi uno sforzo consapevole e sistematico volto a influenzare le opinioni e le azioni di un pubblico o di un'intera società. In questo contesto, il termine 'propaganda' viene originariamente utilizzato dalla Chiesa cattolica per indicare attività di proselitismo, senza connotazioni esplicitamente negative."

Successivamente, è stato ampliato lo studio sulla persuasione e le tecniche di propaganda, approfondendo la lettura di testi specifici.

Lo studio condotto sulla propaganda si basa principalmente su due lavori: **'The Science of Persuasion'** di Robert B. Cialdini e **'A Survey on Computational Propaganda Detection'** di G. Da San Martino, S. Cresci, A. Barron-Cede, S. Yu, R. Di Pietro e P. Nakov.

Il primo paper esplicita come la persuasione sia la base della propaganda e si fonda su sei principi chiave:

- **Consistency:**

Questo principio si basa sull'idea che le persone tendono a comportarsi in modo coerente con le loro precedenti affermazioni o impegni. Una volta che qualcuno ha preso una posizione o ha fatto una promessa, è più probabile che agisca in modo coerente con quella scelta per mantenere una buona immagine di sé.

- **Validazione sociale:**

Le persone tendono a seguire cio' che vedono fare dagli altri.

Se molte persone fanno qualcosa, e' facile pensare che sia la cosa giusta o accettabile. Tuttavia, se questo principio viene usato male, ad esempio in una campagna che promuove comportamenti negativi, puo' portare a un aumento dell'accettazione di tali comportamenti, proprio perche' altre persone li praticano.

- **Affinita':**

Le persone sono piu' inclini a dire di si' a chi gli piace. Quando ce' una connessione personale o un'affinita', e' piu' facile che una persona accetti una richiesta. Complimenti sinceri e interazioni positive possono rafforzare questo legame, aumentando la predisposizione ad ascoltare e a rispondere favorevolmente.

- **Autorita':**

Le persone tendono a seguire i consigli di chi percepiscono come un'autorita' o un esperto. L'autorita' puo' essere riconosciuta da titoli, dall'abbigliamento o dal comportamento di una persona, e questo influisce molto sulle decisioni e le azioni degli altri.

- **Scarsita':**

Quando qualcosa e' raro o difficile da ottenere, sembra piu' prezioso. La percezione di scarsita' aumenta il desiderio di quella cosa. Questo principio e' spesso usato nel marketing per creare un senso di urgenza e spingere i consumatori a competere per un'opportunita' limitata.

- **Conoscenza:**

La conoscenza e' potere. Sapere come funzionano i principi di persuasione puo' dare un vantaggio nelle interazioni sociali e nelle decisioni quotidiane. Chi sa come usare queste tecniche in modo efficace puo' ottenere risultati migliori, mentre chi non le conosce puo' trovarsi in difficolta'.

Il secondo paper, *A Survey on Computational Propaganda Detection*, ha offerto al team una panoramica esaustiva sulle tecniche di rilevamento della propaganda computazionale, attraverso l'importanza di un approccio integrato che combini **l'analisi del linguaggio naturale (NLP)** e **l'analisi delle reti (Network Analysis)**.

Le tecniche di rilevamento della propaganda si basano principalmente sull'analisi testuale, sfruttando strumenti avanzati di NLP per identificare e classificare le tecniche propagandistiche presenti nei testi.

Tra le principali attivita' spiccano la classificazione binaria, che determina se una frase contiene elementi propagandistici, e la classificazione multi-etichetta, che identifica frammenti testuali specifici utilizzando determinate tecniche e categorizzandoli.

Modelli contestuali come **BERT** e **RoBERTa**, utilizzati all'interno del codice del progetto, si sono dimostrati particolarmente efficaci nel migliorare la precisione e l'affidabilita' della classificazione.

Un'ulteriore studio sulla propaganda e la narrativa e' stato effettuato tramite il paper **Reports of personal experiences and stories in argumentation: Datasets and Analysis** di N. Faalk e G. Lapesa, Institute for Natural Language Processing, University of Stuttgart.

Il paper analizza la narrativa come un elemento essenziale dell'argomentazione e della discussione pubblica, ponendo particolare attenzione ai racconti personali e alle storie condivise dalle persone. Queste narrazioni, spesso basate su esperienze individuali e arricchite da elementi emotivi, si rivelano strumenti potenti per catturare l'attenzione, favorire l'empatia e rendere gli argomenti pi accessibili e relazionabili.

La ricerca ha sottolineato come le narrazioni personali possano migliorare la qualita' del dibattito, aumentando la persuasivita' degli argomenti e facilitando la comprensione reciproca tra i partecipanti.

In contesti complessi, come le discussioni su questioni sociali, le sole informazioni fattuali potrebbero non essere sufficienti a comunicare l'urgenza o l'importanza di una tematica. Le storie, in questi casi, diventano un mezzo indispensabile per costruire un dialogo piu' profondo e consapevole.

Queste conoscenze hanno guidato il team nello sviluppo del codice, assicurando che le soluzioni implementate tenessero conto delle dinamiche narrative e delle loro caratteristiche, per un'analisi piu' accurata e sensibile del contenuto testuale.

1.3 Organizzazione del gruppo

Il progetto si e' svolto nell'arco di 20 giorni, con un impegno medio di circa 5 ore al giorno, seguendo una struttura collaborativa che ha ottimizzato l'assegnazione dei compiti.

Bruno Maria Di Maio ha iniziato immediatamente a lavorare sui primi due punti di scraping, concentrandosi sull'aggiunta di informazioni sull'autore e dei metadati relativi ai discorsi. Lucia Brando, cooperando con Denise Brancaccio, si e' occupata dei punti successivi, con un'attenzione particolare alla definizione e selezione delle feature da includere nel dataset.

Successivamente, Bruno Maria Di Maio ha integrato e ottimizzato il codice, gestendo la creazione degli offset e implementando modelli di linguaggio per il trattamento avanzato del testo.

Infine Denise Brancaccio e Lucia Brando, hanno curato la redazione della documentazione finale, sintetizzando le scelte metodologiche e le fasi operative.

Questa suddivisione ha garantito un'organizzazione efficace e ha permesso di completare il progetto nei tempi previsti, soddisfacendo i requisiti richiesti dalla traccia.

1.4 Tecnologie utilizzate

Il codice e' stato sviluppato utilizzando **Python 3.12.7** come versione di riferimento. Per lo sviluppo e' stato utilizzato l'ambiente **VSCode**, configurato con i requisiti specifici del progetto.

E' stata inoltre integrata un'intelligenza artificiale basata sui modelli di OpenAI per svolgere alcune sezioni del progetto. L'accesso ai servizi di OpenAI avviene tramite una chiave API dedicata, memorizzata in un file locale per garantire la sicurezza, caricata nel seguente modo:

```
os.environ["OPENAI_API_KEY"] = open("src/api key/API-GLHF.txt", "r").read()
```

Le librerie utilizzate sono le seguenti:

- **os**: serve per interagire con il sistema operativo, permettendo di gestire file, directory, variabili d'ambiente e processi.
- **pandas**: e' utilizzata per analisi e manipolazione dei dati, offrendo strutture come DataFrame e Series per lavorare con dati tabellari.
- **openai**: e' una libreria per interagire con le API di OpenAI, usata per implementare modelli di intelligenza artificiale come ChatGPT e DALL-E.
- **concurrent.features**: serve per gestire l'esecuzione parallela di attivita', facilitando l'utilizzo di thread o processi per operazioni asincrone.
- **itertools**: offre strumenti per lavorare con iteratori, come combinazioni, permutazioni e generatori infiniti, utili per ottimizzare algoritmi.

- **transformers**: e' una libreria di Hugging Face per utilizzare modelli di machine learning avanzati (es. BERT, GPT) in compiti di NLP e altro.
- **time**: fornisce funzioni per lavorare con il tempo, come misurare la durata, formattare date o gestire ritardi temporali.
- **threading**: consente di creare e gestire thread in Python, utili per eseguire piu' operazioni simultaneamente in un'applicazione.
- **torch**: libreria principale di PyTorch, usata per il machine learning, con strumenti per creare e addestrare reti neurali.
- **nlk**: libreria per l'elaborazione del linguaggio naturale che include strumenti per analisi testuali, tokenizzazione, stemming, analisi grammaticale e creazione di modelli linguistici.
- **sklearn**: libreria per il machine learning che offre strumenti per classificazione, regressione, clustering e preprocessing dei dati, con un'interfaccia coerente e modulare.
- **wikipedia**: libreria Python che consente di interagire con l'API di Wikipedia per estrarre informazioni, riassunti, contenuti delle pagine e link correlati.
- **requests**: libreria per effettuare richieste HTTP in modo semplice e intuitivo. Supporta GET, POST, autenticazione, gestione di sessioni e manipolazione di header.
- **bs4** (*Beautiful Soup*): libreria per il parsing di documenti HTML e XML, progettata per estrarre dati strutturati da pagine web. Utile per il web scraping.
- **collections**: modulo della libreria standard Python che offre tipi di dati specializzati come Counter, defaultdict, deque e OrderedDict per la gestione efficiente di collezioni di dati.
- **re**: modulo della libreria standard Python per la gestione delle espressioni regolari, utilizzato per ricerca, sostituzione e analisi avanzata di stringhe.
- **textstat**: libreria per l'analisi della leggibilita' della complessita' dei testi. Fornisce metriche come l'indice di Flesch-Kincaid, SMOG, Gunning Fog e altri punteggi di leggibilita'.
- **textblob**: libreria per l'elaborazione del linguaggio naturale che include strumenti per analisi del sentiment, traduzione, correzione grammaticale e analisi testuali semplificate.
- **spacy**: libreria avanzata per l'elaborazione del linguaggio naturale, ottimizzata per grandi dataset e applicazioni di deep learning. Include strumenti per tokenizzazione, analisi sintattica, riconoscimento di entit e embedding.
- **typing**: modulo della libreria standard Python per fornire annotazioni di tipo statico, inclusi tipi complessi come List, Tuple, Dict, Union e altro, per una programmazione pi leggibile e sicura.

2 Descrizione del codice

Il codice sviluppato prende in input un file TSV e rinomina le colonne in surname, code e speech per uniformare i dati.

Successivamente, sfruttando il modello Llama, raccoglie informazioni aggiuntive come fullname, birthday, birthplace, death day, death place, political party, speech date, speech location, speech occasion, topic, cognitive bias e produce il riassunto dello speech.

Inoltre, genera analisi avanzate come la narrativa di sottofondo, il contesto, la struttura retorica e le key-words.

Parallelamente, una serie di modelli pre-addestrati viene utilizzata per analisi specifiche: sentiment analysis, emotion analysis, propaganda detection (incluso il tipo di propaganda e la localizzazione degli offset), toxicity classification, persuasion analysis, TF-IDF, narrative structure analysis, readability, formality e word embedding. Infine, i risultati vengono visualizzati graficamente per facilitare l'interpretazione delle relazioni e delle caratteristiche del testo.

2.1 Modelli

Nel progetto sono stati utilizzati modelli pre-addestrati disponibili sulla piattaforma Hugging Face che hanno permesso al team di effettuare analisi complesse come la classificazione del testo, il riconoscimento di entità e l'analisi delle narrazioni propagandistiche.

I modelli sono stati caricati attraverso la seguente funzione:

```
1 def load_model_and_tokenizer(model_name, device=None):
2     tokenizer = AutoTokenizer.from_pretrained(model_name)
3     model = AutoModelForSequenceClassification.from_pretrained(model_name)
4
5     if device and device != "cpu":
6         model = model.to(device)
7
8     return tokenizer, model
```

Per evitare deadlock e' stato disattivato il parallelismo tra i tokenizers.

```
1 os.environ["TOKENIZERS_PARALLELISM"] = "false"
```

Per gestire il limite di 512 caratteri e' stata implementata una funzione che suddivide automaticamente i testi piu' lunghi in segmenti compatibili con il modello, assicurando che ogni frammento sia elaborato correttamente.

Successivamente, i risultati parziali vengono aggregati per fornire un'analisi completa e coerente del testo originale, garantendo così la gestione efficace di documenti di grandi dimensioni.

```
1 # Funzione per creare chunk con finestra scorrevole
2 def split_text_sliding_window(text, chunk_size=512, overlap=128):
3     words = text.split()
4     chunks = []
5     start = 0
6
7     while start < len(words):
8         end = start + chunk_size
9         chunk = ' '.join(words[start:end])
10        chunks.append(chunk)
11        start = end - overlap
12
13    return chunks
```

2.1.1 Sentiment Analysis

```
1 sentiment_tokenizer, sentiment_model = load_model_and_tokenizer("nlptown/bert-base-  
2     multilingual-uncased-sentiment")  
3 sentiment_labels = ["very negative", "negative", "neutral", "positive", "very  
    positive"]
```

2.1.2 Emotion Analysis

```
1 emotion_tokenizer, emotion_model =  
    load_model_and_tokenizer("bhadresh-savani/distilbert-  
2     base-uncased-emotion")  
3 emotion_labels = ["sadness", "joy", "love", "anger", "fear", "surprise"]
```

2.1.3 Propaganda Classifier

```
1 propaganda_classifier = pipeline("text-classification",  
    model="IDA-SERICS/PropagandaDetection", device=0 if device != "cpu" else -1)
```

2.1.4 Toxicity Classifier

```
1 toxicity_classifier = pipeline("text-classification",  
    model="citizenlab/distilbert-base-multilingual-cased-toxicity", device=0 if  
    device != "cpu" else -1)
```

2.1.5 Word Embedding

```
1 tokenizer = AutoTokenizer.from_pretrained("sentence-transformers/  
2     all-MiniLM-L6-v2")  
3 model = AutoModel.from_pretrained("sentence-transformers/  
4     all-MiniLM-L6-v2")
```


3 Sequence Diagram

3.1 Funzione def speech_info(politician,speech)

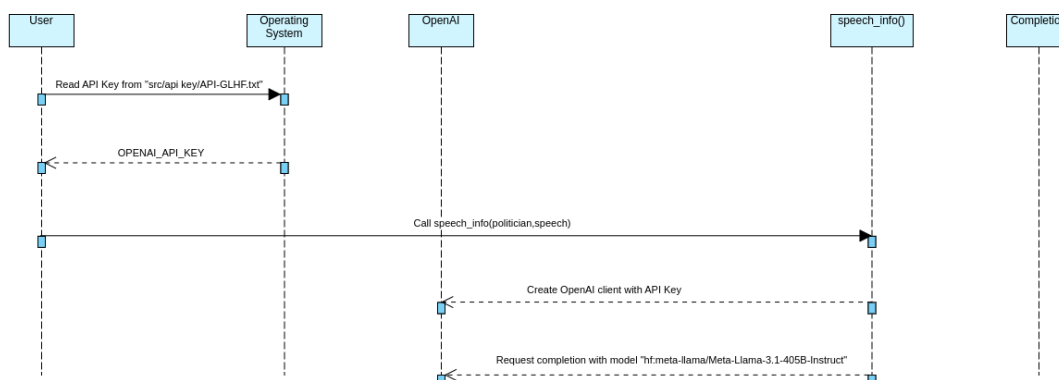
Il diagramma illustra un flusso di interazioni tra quattro attori principali: User, Operating System, OpenAI e Completion.

L'utente avvia il processo.

Il sistema operativo legge l'API Key necessaria per interagire con i servizi OpenAI. Questa chiave memorizzata localmente in un file specifico ("src/api key/API-GLHF.XE"). L'utente riceve l'API Key dall'Operating System e chiama una funzione denominata "speech.info", passando come parametri "politician" e "speech".

L'Operating System crea un client OpenAI utilizzando l'API Key ottenuta in precedenza. Questo client servirà per comunicare con i servizi di OpenAI. Il client OpenAI invia una richiesta di completamento al modello di linguaggio "ht meta-liama/Meta-Llama-3.1-4058-instruct".

Il modello di linguaggio genera il testo completato e lo restituisce al client OpenAI.

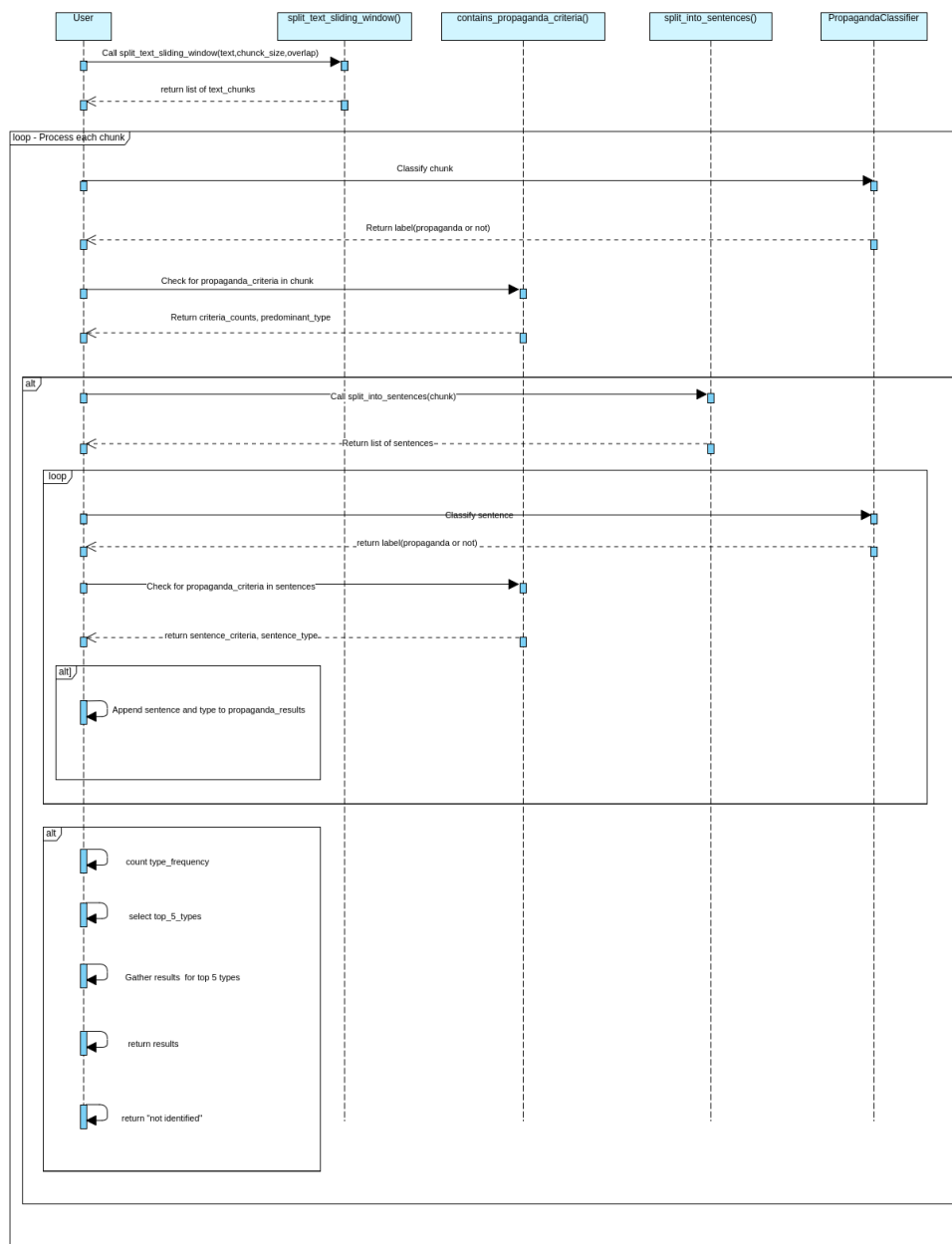


3.2 Funzione def detect_propaganda_type(text, classifier=propaganda_classifier, chunk_size=512, overlap=128)

Il diagramma descrive l'interazione sequenziale tra diversi oggetti all'interno di un sistema di analisi del testo. Il processo ha inizio con una chiamata a una funzione che suddivide un testo in finestre scorrevoli (sliding windows). Ciascuna finestra (chunk) viene processata individualmente. Per ogni chunk:

- Viene eseguita una classificazione iniziale.
- Il chunk viene suddiviso in frasi.
- Si effettua un controllo sulla presenza di determinate parole o pattern (probabilmente legati alla propaganda).
- Vengono conteggiate le frequenze di alcune tipologie di parole o frasi.
- Vengono selezionate le prime 5 tipologie più frequenti.
- Si effettuano ulteriori ricerche o calcoli basati sulle tipologie selezionate.

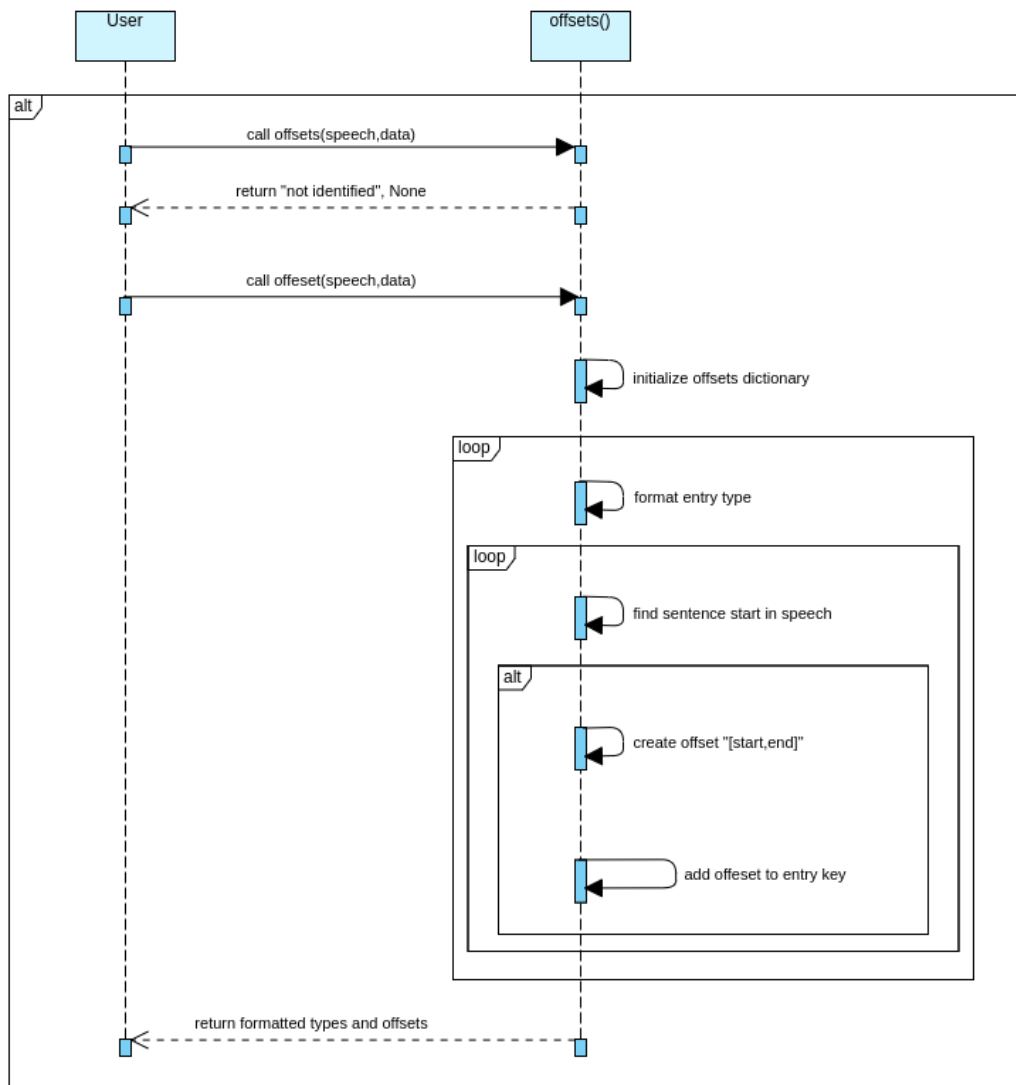
Alla fine di questo processo iterativo, vengono restituiti i risultati dell'analisi, che includono una classificazione complessiva del testo e l'individuazione di specifici pattern propagandistici.



3.3 Funzione def offsets(speech,data)

Il diagramma inizia con l'Utente che interagisce con il sistema. Viene effettuata una chiamata a `offsets(speech data)` che restituisce il valore "not identified" e None nel momento in cui la chiamata non va a buon fine. Viene, dunque, richiamata la funzione `offset()` che va a buon fine; il sistema inizializza un di-

zionario offsets, una struttura dati utilizzata per memorizzare informazioni relative agli offset. Il sistema individua l'inizio della frase all'interno dei dati passati e crea un array di offset con due elementi, "start" e "end". Il sistema, successivamente, aggiunge l'array di offset alla chiave dell'entry, che rappresenta un passaggio nel processo complessivo di gestione dei dati. Infine, la funzione offsets() restituisce i tipi formattati e gli offset.



4 Funzioni

4.1 Verify Politician

La funzione verifica se un dato titolo della pagina web corrisponde a una figura politica. Utilizza il riassunto della prima frase della pagina di Wikipedia e controlla se contiene parole chiave come "who", "politician" o "statesman". Se si verifica un errore di disambiguazione o la pagina non esiste, restituisce False.

```
1 def verify_politician(titolo):
2
3     try:
4         # Prende la prima frase del riassunto della pagina
5         summary = wikipedia.summary(titolo, sentences=1)
6         # Filtra per le parole chiave contenute nella prima frase
7         return any(keyword in summary.lower() for keyword in ["who", "politician",
8             "statesman"])
9
10    except (wikipedia.exceptions.DisambiguationError,
11        wikipedia.exceptions.PageError):
12        return False
```

4.2 Get Politician Info

La funzione effettua una ricerca, verifica se il cognome e' associato a un politico (tramite verify politician), e scarica l'HTML della pagina per analizzarlo con BeautifulSoup. Successivamente, in base al parametro richiesto, estrae e restituisce informazioni come nome, data e luogo di nascita, data e luogo di morte, o partito politico.

Questi dati vengono salvati nel dizionario per eventuali richieste future.

```
1 # Ottiene le informazioni dal politico
2 def get_info(cognome, parametro):
3     # Verifica se il parametro esiste
4     if cognome in politici and parametro in politici[cognome]:
5         return politici[cognome][parametro]
6
7     # Verifica se il cognome esiste e crea il dizionario
8     elif cognome not in politici:
9         politici[cognome] = {}
10
11    # Verifica se il soup non esiste e lo deve creare
12    if "soup" not in politici[cognome]:
13        # Cerca su wikipedia il cognome dato
14        results = wikipedia.search(cognome)
15
16        # Trova l'indice del risultato del politico tramite la funzione
17        # verify_politician
18        index = next((i for i, result in enumerate(results) if
19            verify_politician(result)), None)
20
21    if index is not None:
22        # Prende l'URL della pagina wikipedia
23        url = wikipedia.page(results[index]).url
24
25        # Effettua una richiesta GET alla pagina
```

```

24     response = requests.get(url)
25
26     # Verifica se la richiesta ha avuto successo
27     if response.status_code == 200:
28         # Crea l'oggetto BeautifulSoup per analizzare l'HTML e lo aggiunge
           dizionario
29     politici[cognome]["soup"] = BeautifulSoup(response.text,
           features="html.parser")
30     else:
31         print("Errore nella richiesta della pagina:", response.status_code)
32
33     if "soup" in politici[cognome]:
34         soup = politici[cognome]["soup"]
35         # Trova l'infobox contenente le rows in cui sono presenti diversi dati
36         infobox = soup.find("table", {"class": "infobox"}).find_all("tr")
37
38         # Cerca il nome
39         if parametro == "name":
40             # Estrai il nome completo
41             name = soup.find("div", {"class": "nickname"}).text
42
43             if name:
44                 # Aggiunge il nome al dizionario
45                 politici[cognome]["name"] = name
46                 # Restituisce il nome
47                 return name
48             return "N/A"
49
50         # Cerca la data di nascita
51         elif parametro == "birthday":
52             # Estrae la data di nascita
53             birth_date = soup.find("span", {"class": "bday"}).text
54
55             if birth_date:
56                 # Aggiunge la data di nascita al dizionario
57                 politici[cognome]["birthday"] = birth_date
58                 # Restituisce la data di nascita
59                 return birth_date
60             return "N/A"
61
62         # Cerca il luogo di nascita
63         elif parametro == "birthplace":
64             # Trova il luogo di nascita in caso questo sia un link
65             try:
66                 # Estrae il luogo di nascita
67                 birth_place = next(
68                     # Prende il link del luogo di nascita e lo unisce allo Stato di
                       nascita
69                     ("%s%s" % (row.find("a").text,
                       row.find("a").find_next_sibling(string=True).text)
70                     for row in infobox if "Born" in row.get_text()),
71                     None
72                 )
73             # Trova il luogo di nascita in caso questo sia una scritta semplice
74             except AttributeError:
75                 # Estrae il luogo di nascita
76                 birth_place = next(
77                     # Prende la stringa del luogo di nascita
78                     ("%s" % row.find("br").find_next_sibling(string=True).text
79                     for row in infobox if "Born" in row.get_text()),

```

```

80         None
81     )
82
83     if birth_place:
84         # Aggiunge il luogo di nascita al dizionario
85         politici[cognome]["birthplace"] = birth_place
86         # Restituisce il luogo di nascita
87         return birth_place
88     return "N/A"
89
90 # Cerca la data di morte
91 elif parametro == "deathday":
92     # Estrae la data di morte
93     death_day = next(
94         # Prende lo span della data di morte
95         (row.find("span").text
96          for row in infobox if "Died" in row.get_text()),
97         None
98     )
99
100    if death_day:
101        # Riformatta la data per togliere le parentesi ed avere il formato
102        # aaaa-mm-gg
103        death_day = death_day[1:-1]
104        # Aggiunge la data di morte al dizionario
105        politici[cognome]["deathday"] = death_day
106        # Restituisce la data di morte
107        return death_day
108    # Aggiunge None al dizionario nel caso in cui la persona non sia morta
109    politici[cognome]["deathday"] = None
110    return None
111
112 # Cerca il luogo di morte
113 elif parametro == "deathplace":
114     # Trova il luogo di morte in caso questo sia un link
115     try:
116         # Estrae il luogo di morte
117         death_place = next(
118             # Prende il link del luogo di morte e lo unisce allo Stato di morte
119             ("%s%s" % (row.find("a").text,
120                        row.find("a").find_next_sibling(string=True).text)
121              for row in infobox if "Died" in row.get_text()),
122             None
123         )
124     # Trova il luogo di nascita in caso questo sia una scritta semplice
125     except AttributeError:
126         death_place = next(
127             # Prende la stringa del luogo di morte
128             ("%s" % row.find("br").find_next_sibling(string=True).text
129              for row in infobox if "Died" in row.get_text()),
130             None
131         )
132
133     if death_place:
134         # Aggiunge il luogo di morte al dizionario
135         politici[cognome]["deathplace"] = death_place
136         # Restituisce il luogo di morte
137         return death_place
138     # Aggiunge None al dizionario nel caso in cui la persona non sia morta
139     politici[cognome]["deathplace"] = None

```

```

138         return None
139
140     # Cerca il partito politico
141     elif parametro == "party":
142         # Estrae il partito politico
143         party = next(
144             # Prende il link del partito politico
145             ("%s" % row.find("a").text
146              for row in infobox if "Political party" in row.get_text()),
147             None
148         )
149
150         if party:
151             # Aggiunge il partito politico al dizionario
152             politici[cognome]["party"] = party
153             # Restituisce il partito politico
154             return party
155         return "N/A"
156
157     else:
158         print("Parametro selezionato non valido.")

```

4.3 Speech Info

La funzione utilizza llama per ottenere varie informazioni sul discorso.

Oltre alla data e al luogo in cui si e' tenuto il discorso, attraverso il prompt vengono riportati anche l'occasione in cui si tenuto, il topic di cui tratta, i cognitive bias, una summary e un'analisi dello storytelling.

```

1 def speech_info(politician, speech):
2     try:
3         client = openai.OpenAI(
4             api_key=os.environ.get("OPENAI_API_KEY"),
5             base_url="https://glhf.chat/api/openai/v1"
6         )
7         completion = client.chat.completions.create(
8             model="hf:meta-llama/Meta-Llama-3.1-405B-Instruct"
9             messages=[Prompt]
10            temperature=0
11
12        return speech, completion.choices[0].message.content
13
14    except (openai.RateLimitError, Exception):
15        raise

```

4.4 Speech Data

Questa funzione e' progettata per estrarre i dati associati a un determinato discorso (speech) da un dizionario (results dict) e formattarli in un array con esattamente 8 valori, gestendo eventuali errori.

```

1 def get_speech_data(speech, results_dict):
2     try:
3         parts = results_dict[speech].split('$')
4         parts += ["N/A"] * (8 - len(parts))

```



```

5         return parts[:8]
6     except (KeyError, Exception):
7         return ["N/A", "N/A", "N/A", "N/A", "N/A", "N/A", "N/A", "N/A"]

```

4.5 Classify Text

La funzione classifica un testo in base a un modello pre-addestrato.

```

1 # Funzione unificata per classificare il testo
2 def classify_text(text, model, tokenizer, labels):
3     inputs = tokenizer(text, return_tensors="pt", truncation=True)
4     outputs = model(**inputs)
5     probabilities = torch.nn.functional.softmax(outputs.logits, dim=1)
6     prediction = torch.argmax(probabilities, dim=1).item()
7     return labels[prediction].capitalize()

```

4.6 Split Sentences

La funzione suddivide un testo in singole frasi utilizzando una semplice espressione regolare.

```

1 # Dividi il testo in frasi usando una semplice regex
2 def split_into_sentences(text):
3     sentence_endings = re.compile(r'(?<=[.!?])\s+')
4     return sentence_endings.split(text)

```

4.7 Propaganda Criteria

La funzione analizza un testo per rilevare la presenza di elementi riconducibili a tecniche di propaganda. Confronta il contenuto del testo con un insieme di parole o frasi chiave definite per ciascun criterio, come toni accusatori, distorsione dell'opposizione o appelli alla paura.

Per ogni criterio, conta quante volte le frasi chiave sono presenti nel testo.

Se vengono rilevati uno o più criteri, restituisce un dizionario con il conteggio delle occorrenze e il criterio predominante (quello con più corrispondenze).

Se non viene rilevato nulla, restituisce 'None'.

```

1 # Funzione per rilevare i criteri di propaganda
2 def contains_propaganda_criteria(text):
3     criteria = {
4         "accusatory_tone": ["fault of"],
5         "opposition_distortion": ["enemies", "traitor", "dishonest", "corrupt"],
6         "slogan_repetition": ["you have to believe", "the truth is", "don't
7             forget", "do not forget"],
8         "fear_appeals": ["danger", "threat", "crisis", "chaos"],
9         "flagwaving": ["our nation", "homeland", "defend values"],
10        "black_white_fallacy": ["either with us or against us", "only choice"],
11        "click_s": ["golden times", "like once upon a time", "old times"]
12    }
13     # Conta quante frasi/parole chiave sono presenti per ciascun criterio

```

```

13 type_counts = {key: sum(phrase in text.lower() for phrase in phrases) for key,
14                   phrases in criteria.items()}
15 # Filtra i criteri che hanno almeno una corrispondenza
16 detected_types = {key: count for key, count in type_counts.items() if count >
17                   0}
18 # Se non ci sono criteri rilevati, ritorna None
19 if not detected_types:
20     return None, None
21 predominant_type = max(detected_types, key=detected_types.get)
22 return detected_types, predominant_type

```

4.8 Detect Propaganda Type

La funzione suddivide il testo in chunk sovrapposti per garantire che tutte le parti siano valutate senza perdere contesto.

Ogni chunk viene analizzato sia tramite un modello di classificazione di propaganda che con una funzione che cerca criteri definiti di propaganda nel testo.

Se un chunk e' classificato come propaganda o contiene criteri rilevanti, viene ulteriormente diviso in frasi per un'analisi pi' dettagliata.

Per ciascuna frase classificata come propaganda, vengono salvati il tipo di propaganda identificato e la frase stessa.

Infine, se vengono rilevati risultati, la funzione restituisce i cinque tipi di propaganda piu' comuni, ciascuno accompagnato dalle frasi associate.

Se non viene rilevato nulla, restituisce "Not identified".

```

1 def detect_propaganda_type(text, classifier=propaganda_classifier, chunk_size=512,
2   overlap=128):
3     # Divide il testo in chunk
4     text_chunks = split_text_sliding_window(text, chunk_size, overlap)
5     propaganda_results = []
6
7     for chunk in text_chunks:
8         # Classifica il chunk
9         is_propaganda_by_model = classifier(chunk, truncation=True,
10         max_length=chunk_size)[0]['label'] == "propaganda"
11         criteria_counts, predominant_type = contains_propaganda_criteria(chunk)
12
13         # Analizza le frasi del chunk
14         if is_propaganda_by_model or criteria_counts:
15             sentences = split_into_sentences(chunk) # Dividi il chunk in frasi
16             for sentence in sentences:
17                 # Classifica la singola frase
18                 sentence_is_propaganda = classifier(sentence,
19                 truncation=True)[0]['label'] == "propaganda"
20                 sentence_criteria, sentence_type =
21                 contains_propaganda_criteria(sentence)
22
23                 if sentence_is_propaganda or sentence_criteria:
24
25                     propaganda_results.append({
26                         "type": sentence_type,
27                         "sentence": sentence
28                     })
29
30     # Se esistono risultati di propaganda

```

```

27 if propaganda_results:
28     # Conta la frequenza di ogni tipo di propaganda
29     types = [result["type"] for result in propaganda_results]
30     type_counts = Counter(types)
31     top_5_types = type_counts.most_common(5)
32     # Estrai i risultati per le 5 tipologie piu comuni
33     results = []
34     for propaganda_type, _ in top_5_types:
35         sentences_of_type = [
36             {"sentence": result["sentence"]}
37             for result in propaganda_results if result["type"] == propaganda_type
38         ]
39         results.append({
40             "type": propaganda_type,
41             "sentences": sentences_of_type
42         })
43
44     return results
45
46     return "Not identified"

```

4.9 Contains Toxicity Criteria

La funzione analizza un testo per identificare criteri di tossicità basati su un insieme di categorie predefinite. Ogni categoria, come insulti, minacce o linguaggio violento, viene associata a una lista di parole o frasi chiave indicative di quella forma di tossicità.

Confronta il testo con le chiavi, calcola il numero di corrispondenze per ciascuna categoria, e restituisce un dizionario con i criteri rilevati e il loro conteggio.

Inoltre, identifica quale categoria è predominante, ovvero quella con il maggior numero di corrispondenze. Se nessun criterio viene rilevato, restituisce None.

```

1 # Funzione per rilevare i criteri di tossicità
2 def contains_toxicity_criteria(text):
3     criteria = {
4         "insults": ["idiot", "stupid", "moron", "fool", "scum", "loser",
5                    "crooked", "degenerate", "ignoramus"],
6         "threats": ["destroy", "kill", "eliminate", "eradicate", "you'll regret",
7                    "annihilate", "purge", "defeat utterly"],
8         "violent_language": ["blood", "war", "fight", "smash", "knock them out",
9                              "battle", "no quarter"],
10        "divisive_statements": ["us vs them", "enemy of the people", "deep state",
11                                "traitors", "fifth column"],
12        "personal_attacks": ["you always fail", "your fault", "worthless",
13                             "incompetent", "phony", "betrayal"],
14        "exaggerations": ["always lie", "never truthful", "the worst", "greatest
15                           challenge", "unprecedented threat"],
16        "fear_appeals": ["danger", "threat", "crisis", "chaos", "they're coming
17                          for you", "crime wave"],
18        "flagwaving": ["our nation", "homeland", "defend values", "make
19                       america great again", "fatherland", "motherland", "maga"]
20    }
21
22    # Conta quante frasi/parole chiave sono presenti per ciascun criterio
23    type_counts = {key: sum(phrase in text.lower() for phrase in phrases) for key,
24                      phrases in criteria.items()}
25
26    # Filtra i criteri che hanno almeno una corrispondenza

```

```

16 detected_types = {key: count for key, count in type_counts.items() if count >
17                     0}
18 # Se non ci sono criteri rilevati, ritorna None
19 if not detected_types:
20     return None, None
21 # Identifica il tipo di tossicità
22 predominant_type = max(detected_types, key=detected_types.get)
23 return detected_types, predominant_type

```

4.10 Classify Toxicity

La funzione classifica un testo identificando eventuali criteri di tossicità.

Utilizza la funzione `contains_toxicity_criteria` per analizzare il testo e determinare quali categorie di tossicità sono presenti, insieme al conteggio delle loro occorrenze.

Se vengono rilevati criteri, questi vengono formattati come una lista leggibile in cui ogni riga mostra il nome del criterio (con spazi al posto degli underscore) e il numero di volte che è stato riscontrato.

```

1 # Funzione per classificare la tossicità
2 def classify_toxicity(text):
3     try:
4         # Rilevamento criteri
5         detected_criteria, _ = contains_toxicity_criteria(text)
6
7         # Formattare i criteri rilevati uno per riga
8         if detected_criteria:
9             return '\n'.join(["%s: %s" % (k, v) for k, v in
10                                detected_criteria.items()]).replace("_", " ").title()
11         else:
12             return ""
13     except Exception:
14         raise

```

4.11 Offsets

La funzione calcola gli offset delle frasi identificate come propaganda all'interno di un discorso, fornendo la posizione iniziale e finale di ogni frase rilevata per ciascun tipo di propaganda attraverso il formato `[inizio:fine]`.

Se il parametro `data` contiene il valore "Not identified", la funzione restituisce direttamente questo valore con `None`, indicando che non sono state rilevate frasi di propaganda.

```

1 def offsets(speech, data):
2
3     # Verifica che la propaganda sia stata correttamente identificata
4     if data == "Not identified":
5         return "Not identified", None
6
7     # Crea un dizionario per definire gli offset
8     offsets = {
9         # Crea una chiave per tipo e la formatta correttamente
10         entry["type"].replace("_", " ").capitalize(): [
11             # Crea un offset per ogni frase
12             "[%s:%s]" % (start, start + len(sentence["sentence"]))

```

```

13         for sentence in entry["sentences"]
14             # Trova le frasi e le misura
15             if (start := speech.find(sentence["sentence"])) != -1
16         ]
17     for entry in data
18 }
19
20 # Restituisce il tipo di propaganda e gli offset per ogni tipo di propaganda
21 return "\n".join(offsets.keys()), "\n".join("%s: %s" % (key, ",
    ".join(element)) for key, element in offsets.items())

```

4.12 Formality

La funzione determina se un discorso e' formale o informale in base al suo livello di soggettivita' utilizzando il modulo TextBlob per analizzare il sentiment del testo.

Se la lunghezza del discorso e' inferiore o uguale a 150 caratteri, la funzione restituisce "N/A", indicando che la valutazione non e' applicabile.

Se il discorso e' piu' lungo, la funzione analizza la soggettivita' del testo: una soggettivita' inferiore a 0,5 suggerisce che il testo e' formale, mentre una soggettivita' piu' alta indica che il discorso e' informale.

```

1 def is_formal(speech):
2     if len(speech) <= 150:
3         return "N/A"
4
5     blob = TextBlob(speech)
6     # Bassa soggettivita indica formalita
7     if blob.sentiment.subjectivity < 0.5:
8         return "Formal"
9     else:
10        return "Informal"

```

4.13 Readability

La funzione calcola diverse metriche di leggibilita' di un discorso utilizzando il modulo textstat, che fornisce diversi indici per valutare quanto sia facile o difficile leggere un testo.

Le metriche prese in considerazione sono: *flesch reading ease*, *flesch-kincaid grade*, *gunning fog*, *SMOG index*.

```

1 # Funzione per calcolare metriche di leggibilita
2 def calcola_legibilita(speech):
3     return {
4         "flesch_reading_ease": textstat.flesch_reading_ease(speech), # facilita di
5             lettura basandosi sulla lunghezza delle frasi e sul numero di sillabe
6             per parola
7         "flesch_kincaid_grade": textstat.flesch_kincaid_grade(speech), # anni di
8             istruzione necessari per comprendere il testo
9         "gunning_fog": textstat.gunning_fog(speech), # complessita del speech in
10             base alla lunghezza delle frasi e alla percentuale di parole complesse
11         "smog_index": textstat.smog_index(speech), # come quello precedente
12         "text_standard": textstat.text_standard(speech) # sintetizzazione dei
13             risultati in un livello scolastico approssimativo
14     }

```

4.14 Analyze Narrative Structure

La funzione analizza la struttura narrativa di un discorso, concentrandosi su vari aspetti legati alla lunghezza e alla variabilità delle frasi.

In seguito alla tokenizzazione, viene calcolato il numero di parole all'interno di ogni token, generando una lista che rappresenta la lunghezza di ogni frase.

Dopo aver ottenuto le lunghezze delle frasi, la funzione calcola diverse metriche narrative.

La lunghezza media delle frasi viene determinata calcolando la media aritmetica del numero di parole per ciascuna frase.

Viene poi calcolata la deviazione standard, che misura quanto le lunghezze delle frasi variano l'una dall'altra, dando un'idea della coerenza o della varietà nella struttura del discorso.

Inoltre, la funzione conta il numero totale di frasi nel discorso e, infine, calcola un punteggio di complessità, che è il rapporto tra la lunghezza media delle frasi e la deviazione standard.

```
1 def analyze_narrative_structure(speech):
2     sentences = sent_tokenize(speech)
3
4     # Analyze sentence length variation
5     sentence_lengths = [len(sentence.split()) for sentence in sentences]
6
7     narrative_metrics = {
8         'Average Sentence Length': np.mean(sentence_lengths),
9         'Sentence Length Variation': np.std(sentence_lengths),
10        'Total Sentences': len(sentences),
11        'Complexity Score': np.mean(sentence_lengths) / (np.std(sentence_lengths)
12        + 1)
13    }
14
15    return "\n".join(["%s: %.3f" % (key, value) if isinstance(value, float) else
16        "%s: %d" % (key, value) for key, value in narrative_metrics.items()])
```

4.15 TF-IDF

La funzione estrae le parole chiave più rilevanti da un discorso utilizzando il metodo TF-IDF.

Imposta un limite sul numero di parole chiave da restituire (top 5) e ignora le stop words.

Successivamente, calcola la matrice TF-IDF e seleziona le parole con i punteggi più alti. Le parole chiave e i loro punteggi vengono quindi ordinati e restituiti in una stringa formattata con i punteggi arrotondati a tre decimali, mettendo in evidenza le parole più significative nel testo.

```
1 def tfidf(speech):
2     # Definisce il numero di parole massimo per le keywords
3     max_ngram_length = 1
4     # Definisce quante keywords dobbiamo trovare
5     top_n = 5
6
7     # Inserisce i parametri di ricerca e il dizionario delle stop words
8     vectorizer = TfidfVectorizer(
9         ngram_range=(1, max_ngram_length),
```

```

10     stop_words='english'
11 )
12
13 # Prende la matrice TF-IDF
14 tfidf_matrix = vectorizer.fit_transform([speech])
15
16 # Prende i nomi delle parole
17 feature_names = vectorizer.get_feature_names_out()
18
19 # Prende gli score TF-IDF
20 tfidf_scores = tfidf_matrix.toarray()[0]
21
22 # Crea una lista di tuple (keyword, score)
23 keyword_scores = [
24     (feature_names[idx], score)
25     for idx, score in enumerate(tfidf_scores)
26     if score > 0
27 ]
28
29 # Ordina per score decrescente
30 keyword_scores = sorted(keyword_scores, key=lambda x: x[1], reverse=True)
31
32 # Prende solo le top_n keywords
33 top_keywords = keyword_scores[:top_n]
34
35 # Restituisce le parole chiave con i rispettivi punteggi
36 return "\n".join(["%s: %.3f" % (kw.title(), round(score, 3)) for kw, score in
    top_keywords])

```

4.16 Word Embeddings

La funzione calcola gli embeddings di un testo suddividendolo in chunk di dimensione specificata, con un sovrapposizione tra i chunk.

Per ogni chunk, il testo viene tokenizzato e passato attraverso il modello per ottenere gli embeddings dei token.

Gli embeddings di ogni token nel chunk vengono mediati (calcolando la media dei vettori) per ottenere un embedding complessivo per il chunk.

Alla fine, gli embeddings medi di tutti i chunk vengono mediati nuovamente per ottenere un embedding finale che rappresenta l'intero testo.

```
1 # Funzione per calcolare gli embeddings sui chunk
2 def calcola_embedding(testo, tokenizer, model, chunk_size=512, overlap=128):
3     chunks = split_text_sliding_window(testo, chunk_size=chunk_size,
4         overlap=overlap)
5     embeddings_chunk = []
6
7     for chunk in chunks:
8         inputs = tokenizer(chunk, return_tensors="pt", padding=True,
9             truncation=True, max_length=chunk_size)
10
11         with torch.no_grad():
12             outputs = model(**inputs)
13
14             token_embeddings = outputs.last_hidden_state
15             mean_embedding = token_embeddings.mean(dim=1).squeeze()
16             embeddings_chunk.append(mean_embedding.cpu().numpy())
17
18     if embeddings_chunk:
19         final_embedding = np.mean(embeddings_chunk, axis=0)
20     else:
21         final_embedding = np.zeros(model.config.hidden_size)
22
23     return final_embedding
```

4.17 Visualize Embeddings

La funzione utilizza i word embedding per creare uno scatter plot in cui ogni punto rappresenti un discorso e ogni colore un politico. Discorsi semanticamente simili saranno altrettanto vicini sul grafico, discorsi semanticamente differenti verranno rappresentati piu' distanti.

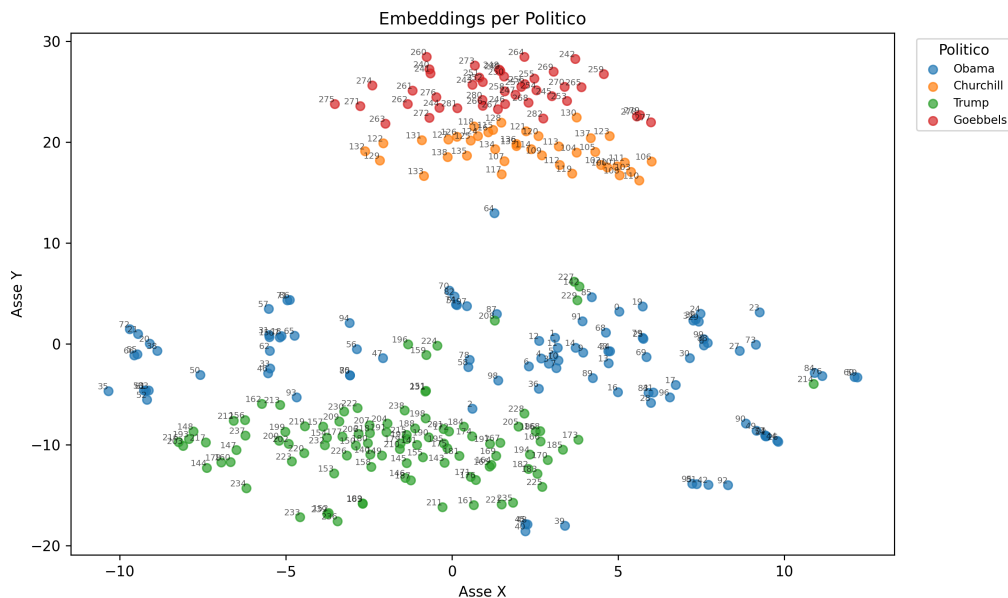
```
1 # Visualizza gli embeddings su scatter plot
2 def visualize_embeddings(dataset, plt_show, output_path="word_embedding.png"):
3     tokenizer = AutoTokenizer.from_pretrained("sentence-transformers/
4         all-MiniLM-L6-v2")
5     model = AutoModel.from_pretrained("sentence-transformers/
6         all-MiniLM-L6-v2")
7
8     dataset["Embedding"] = dataset["Speech"].fillna("").apply(lambda x:
9         calcola_embedding(x, tokenizer, model))
10    embeddings = np.stack(dataset["Embedding"].values)
11
12    tsne = TSNE(n_components=2, random_state=42, perplexity=30)
13    embeddings_2d = tsne.fit_transform(embeddings)
14
15    dataset["x"] = embeddings_2d[:, 0]
16    dataset["y"] = embeddings_2d[:, 1]
17
18    politici_unici = dataset["Surname"].unique()
19
20    # Crea una figura di dimensioni (10, 6) e posiziona i punti in base ai nomi
21    plt.figure(figsize=(10, 6))
```



```

21     for politico in politici_unici:
22         subset = dataset[dataset["Surname"] == politico]
23         plt.scatter(subset["x"], subset["y"], label=politico, alpha=0.7)
24
25     # Enumera ogni discorso sui punti
26     for i, (x, y) in enumerate(zip(dataset["x"], dataset["y"])):
27         plt.text(x, y, str(i), fontsize=6, ha='right', va='bottom', color='black',
28                 alpha=0.6)
29
30     # Impostazioni del grafico (titolo, label di ascissa e ordinate, leggenda e
31     # layout finale)
32     plt.title("Embeddings per Politico", fontsize=12)
33     plt.xlabel("Asse X", fontsize=10)
34     plt.ylabel("Asse Y", fontsize=10)
35     plt.legend(title="Politico", bbox_to_anchor=(1.02, 1), loc="upper left",
36              fontsize=8)
37     plt.tight_layout()
38     # Salva il grafico nella stessa directory del file Python
39     plt.savefig(output_path, dpi=300, bbox_inches='tight')
40     # Chiudi il plot per evitare conflitti in future visualizzazioni
41
42     if plt_show == "s":
43         return plt
44
45     plt.close()
46     return None

```



4.18 Analyze Persuasion

La funzione analizza un testo per valutare la presenza di elementi persuasivi.

Usa spaCy per elaborare il testo e calcolare il numero di superlativi, domande retoriche, e parole emotive,

di urgenza, autorità e prova sociale.

Ogni metrica è normalizzata in base alla frequenza nel testo e ponderata con un peso specifico che riflette l'importanza persuasiva di ciascun elemento.

Somma gli score normalizzati per ottenere un punteggio totale di persuasione.

I risultati vengono restituiti in un formato leggibile, con ogni metrica e il suo valore separati da una riga.

```
1 def analyze_persuasion(text):
2     # Analisi del testo usando spaCy
3     doc = nlp(text.lower())
4
5     # Calcola metriche persuasive
6     # Per ogni tipo di elemento persuasivo, conta quante volte appare nel testo
7     metrics = {
8         # conta i superlativi
9         "superlatives": sum(1 for token in doc if token.tag_ in ["JJS",
10         "RBS"]),
11         # conta le domande retoriche, basandosi sul fatto che terminano con un
12         # punto di domanda
13         "rhetorical_questions": sum(1 for sent in doc.sents if
14         sent.text.strip().endswith("?")),
15         # conta le parole emotive che appartengono alla lista predefinita
16         "emotional_words": sum(1 for token in doc if token.text in
17         PERSUASIVE_WORDS["emotional"]),
18         # conta le parole legate all'urgenza dalla lista predefinita
19         "urgency_words": sum(1 for token in doc if token.text in
20         PERSUASIVE_WORDS["urgency"]),
21         # conta le parole legate all'autorità
22         "authority_words": sum(1 for token in doc if token.text in
23         PERSUASIVE_WORDS["authority"]),
24         # conta le parole di prova sociale
25         "social_proof_words": sum(1 for token in doc if token.text in
26         PERSUASIVE_WORDS["social_proof"])
27     }
28
29     # Calcola il numero totale di token
30     total_tokens = len(doc)
31     # Se il testo non contiene token, restituisce un dizionario vuoto
32     if total_tokens == 0:
33         return {key: 0.0 for key in metrics.keys() | {"total_score"}}
34
35     # Definisce il peso per ogni elemento persuasivo, indicando quanto importante
36     # per ciascuna metrica
37     weights = {
38         "superlatives": 1.5, # peso medio
39         "rhetorical_questions": 2.0, # sono più persuasive
40         "emotional_words": 2.5, # sono le più persuasive
41         "urgency_words": 2.0, # sono persuasive
42         "authority_words": 1.8, # peso medio-alto
43         "social_proof_words": 1.7 # peso inferiore
44     }
45
46     # Calcola uno score normalizzato per ogni metrica
47     # Per ogni metrica:
48     # - Divido il conteggio per il numero totale di token (normalizzazione)
49     # - Moltiplico per il peso associato per enfatizzare la sua importanza
50     normalized_scores = {
51         key: (metrics[key] / total_tokens) * weights[key] for key in metrics
52     }
53
54     # Somma tutti gli score normalizzati per ottenere lo score totale di
```

```

47     persuasione
48     total_score = sum(normalized_scores.values())
49     normalized_scores["total_score"] = round(total_score, 3) # arrotondare a 3 -
        scelta comune per rappresentare valori numerici in cui la precisione oltre
        il millesimo non risulta necessaria
    return '\n'.join("%s: %s" % (k.replace('_', ' ').title(), v) for k, v in
        normalized_scores.items()) # restituisce gli score

```

5 Analisi delle Prestazioni

Il processo di analisi ha richiesto complessivamente *18 minuti e 9,04 secondi*.

Questo tempo e' stato suddiviso in diverse fasi:

- scraping dei dati
- estrazione delle informazioni
- analisi approfondita
- creazione di rappresentazioni vettoriali delle parole

La durata complessiva e' influenzata da diversi fattori, tra cui la quantita' di dati da elaborare, la complessita' delle analisi e le risorse computazionali disponibili.

I risultati dell'analisi, inclusi i dati grezzi e le rappresentazioni vettoriali, sono stati salvati in formato strutturato per facilitare ulteriori approfondimenti e sviluppi.

In futuro, si prevede di ottimizzare il processo per ridurre i tempi di esecuzione, ad esempio attraverso l'implementazione di algoritmi piu' efficienti o l'utilizzo di hardware piu' performante.

Inoltre, si valtera' l'opportunita' di espandere l'analisi includendo nuove feature e modelli linguistici piu' avanzati.

```
🔧 Eseguo lo scraping delle informazioni dei politici
✅ Scraping completato
🕒 Tempo di esecuzione dello scraping: 0 ore, 0 minuti, 11.28 secondi

🌐 Ottengo le informazioni sui discorsi
✅ Informazioni ottenute
🕒 Tempo di esecuzione per l'ottenimento delle informazioni: 0 ore, 7 minuti, 2.01 secondi

🧠 Eseguo l'analisi dei discorsi
✅ Analisi completata
🕒 Tempo di esecuzione dell'analisi: 0 ore, 7 minuti, 52.60 secondi

📄 Aggiungo feature text-based
✅ Aggiunta completata
🕒 Tempo di esecuzione dell'analisi: 0 ore, 0 minuti, 5.23 secondi

📊 Visualizzare il grafico del word embedding? (s/n) n
📄 Calcolo il word embedding
✅ Calcolo completato
🕒 Tempo di esecuzione del word embedding: 0 ore, 1 minuti, 23.65 secondi

✅ Grafico del word embedding salvato in src/word_embedding.png
✅ Dataset salvato in src/dataset/speech-b.csv

🕒 Tempo di esecuzione totale: 0 ore, 17 minuti, 58.43 secondi
```

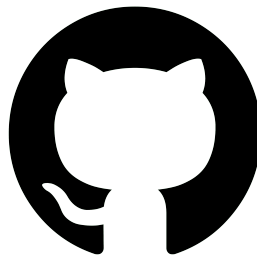
6 Conclusioni

Il team e' riuscito a completare con successo il project work assegnato grazie a una solida collaborazione, una gestione efficace del tempo e un'accurata pianificazione delle attivit.

La continua iterazione, brainstorming e testing dei modelli hanno permesso di superare le difficolta' tecniche iniziali dovute alla novita della richiesta e rispettare i tempi stabiliti.

Alla fine, il lavoro e' stato completato con il raggiungimento degli obiettivi prefissati in modo soddisfacente.

7 Collegamenti esterni



Codice completo:

<https://github.com/lbrando/project-work>