UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Ingegneria Informatica

Elaborato di fine corso : Text Mining

Realizzazione di un ChatBot con tecnologia RAG

Anno Accademico 2023-2024

Studenti:

DI GUIDA SIMONE MARAIA matr. M63001590
MANNA ALESSIA matr. M63001622
MARAIA ALESSANDRA matr. M63001648

Sommario

Realizzazione di un ChatBot con tecnologia RAG I
Introduzione 3
Capitolo 1: Tecnologie utilizzate 4
1.1 RAG 4
1.2 API OpenAI 5
1.3 Anaconda 6
1.4Visual Studio Code 7
1.5 Streamlit 7
1.6 Ollama 8
Capitolo 2: Situazione di partenza del progetto 9
2.1 Fase iniziale 9
2.1.1 bot.py 10
2.1.2 chatbot_streamlit_combined.py 13
2.1.3 falcon.py 16
2.1.4 main_bot.py 17
Capitolo 3 : ParlaMente 20
3.1 Parlamente API OPENAI 21
3.1.1 DataAnalysis.py 21
3.1.2 merger.py 23
3.1.3 OpenAI_utility.py 25
3.1.4 chatbot _parlamente.py 34
3.1.5 Interfaccia 42
3.2 Parlamente : guida all'installazione ed esecuzione con esempi 56

Introduzione

I chatbot assumono ormai un ruolo fondamentale in vari settori, soprattutto nel settore del marketing, nei servizi informativi e nell'ambito di assistenza ai clienti. Questo grazie alla loro capacità di interagire con gli utenti in modo efficace e personalizzato.
Un chatbot è un programma informatico progettato per simulare una conversazione con gli esseri umani utilizzando il linguaggio naturale una delle principali caratteristiche dei chatbot e la loro abilità di interazione linguistica essi sono in grado di comprendere e rispondere al linguaggio umano, permettendo così di risolvere domande fornire informazioni e assistere gli utenti in compiti ben precisi. I chatbot più avanzati utilizzano l'intelligenza artificiale (IA) E tecnologie di machine learning (ML) Per migliorare la comprensione del contesto e delle richieste effettuate dall'utente. È evidente che creare un chatbot sia un compito piuttosto complesso che richiede competenze tecniche avanzate e una buona conoscenza del linguaggio naturale. Bisogna progettare un'architettura solida, sviluppare algoritmi di machine learning, implementare tecnologie di elaborazione del linguaggio ed infine creare un’interfaccia utente user friendly in modo da poter garantire a tutti gli utenti una semplice e veloce comprensione per l'interazione con il chatbot. Per fornire risposte sempre più precise e pertinenti c'è bisogno di migliorare continuamente il sistema e questo lo rende oneroso temporalmente e computazionalmente. Per realizzare il chatbot ParlaMente sono stati utilizzati specifici strumenti e tecnologie che verranno illustrati in seguito fino alla realizzazione finale del progetto.

Capitolo 1: Tecnologie utilizzate

Nel corso dello sviluppo del progetto, sono stati utilizzati svariati strumenti e tecnologie fondamentali, ciascuno dei quali ha svolto un ruolo cruciale nella creazione del chatbot finale. Grazie a questi strumenti, è stato possibile implementare funzionalità avanzate nel chatbot, garantendo un'esperienza utente di alta qualità e un'interazione naturale con l'utente finale.

Durante l'intero processo, per la modellazione e l'analisi del linguaggio naturale, sono state impiegate tecnologie avanzate come i Large Language Model (LLM) e la tecnica RAG (Retrieval Augmented Generation). Queste tecnologie hanno permesso di migliorare notevolmente le capacità del chatbot, rendendolo più efficiente e preciso nelle risposte.

1.1 RAG

La Retrieval Augmented Generation (RAG) è una tecnica avanzata impiegata nei modelli di linguaggio per migliorare la qualità delle risposte generate. Questa tecnica si basa su due componenti principali: il recupero (retrieval) e la generazione (generation).

Nella fase di recupero, il modello cerca e recupera informazioni rilevanti da una vasta base di dati o da un insieme di documenti esterni. Questo passaggio utilizza un modello di recupero, come un modello di embedding, per individuare i documenti o i frammenti di testo più pertinenti alla richiesta dell'utente.

Successivamente, nella fase di generazione, spesso basata su architetture di tipo Transformer, il modello utilizza le informazioni recuperate per costruire una risposta che sia maggiormente pertinente alla domanda.

Questa tecnica si è dimostrata particolarmente utile nella gestione delle conversazioni generative, poiché consente al chatbot di comprendere il contesto della conversazione e di generare risposte coerenti e contestualmente rilevanti.

I Large Language Model (LLM) hanno contribuito, parallelamente, a migliorare significativamente la capacità del chatbot di comprendere e generare testi coerenti e informativi, fornendo una base solida per la creazione di conversazioni molto naturali. L'integrazione di LLM e RAG ha quindi permesso di sviluppare chatbot capaci di avere interazioni linguistiche di alto livello, arricchendo l'esperienza utente con risposte precise e contestuali.

1.2 API OpenAI

La versione iniziale del nostro progetto si avvale delle API di OpenAI, le quali offrono una vasta gamma di servizi e funzionalità per lo sviluppo di chatbot avanzati. Queste API permettono l'accesso a modelli di linguaggio altamente performanti, come il GPT (Generative Pre-trained Transformer). L'utilizzo delle API di OpenAI ha facilitato l'integrazione dei modelli di linguaggio all'interno del nostro chatbot, sfruttando la potenza e l'efficacia di questi modelli senza la necessità di sviluppare complessi algoritmi di machine learning ex novo.

L'integrazione con le API di OpenAI ha portato numerosi vantaggi. Innanzitutto, ha consentito una rapida implementazione delle funzionalità di linguaggio naturale, accelerando notevolmente il processo di sviluppo. Inoltre, l'accesso a modelli pre-

addestrati ha migliorato la qualità delle interazioni con gli utenti, grazie alla capacità del chatbot di comprendere e generare risposte contestuali e pertinenti.

Grazie a questa soluzione, il nostro chatbot può gestire una vasta gamma di richieste e conversazioni in modo naturale e fluido, offrendo un'esperienza utente ottimizzata. Questo approccio ha anche ridotto significativamente i costi e le risorse necessarie per lo sviluppo, permettendoci di concentrare gli sforzi su altri aspetti del progetto, come l'ottimizzazione dell'interfaccia utente e l'integrazione con altre piattaforme e servizi.

In sintesi, l'utilizzo delle API di OpenAI ha rappresentato un elemento cruciale per il successo della prima versione del nostro chatbot, fornendo una solida base tecnologica su cui costruire e migliorare ulteriormente le capacità del sistema.

1.3 Anaconda

Anaconda si è rivelato uno strumento fondamentale per la gestione dell'ambiente di sviluppo e delle dipendenze del progetto. Questo ambiente integrato offre un'ampia gamma di strumenti e librerie per l'analisi dei dati e lo sviluppo di modelli di machine learning, semplificando notevolmente il processo di sviluppo e distribuzione del chatbot.

Grazie ad Anaconda, siamo riusciti a creare un ambiente di sviluppo altamente configurabile e facilmente replicabile, il che ha ridotto significativamente i tempi di configurazione e risoluzione dei problemi legati alle dipendenze. La gestione centralizzata dei pacchetti tramite Conda ha facilitato l'installazione e l'aggiornamento di librerie cruciali, come TensorFlow, PyTorch, Scikit-learn e molte altre.

1.4Visual Studio Code

Visual Studio Code è stato il nostro strumento principale per scrivere il codice e gestire il progetto. Con le numerose estensioni disponibili, ci ha fornito un ambiente di sviluppo completo e personalizzabile. Soprattutto per la programmazione in Python, Visual Studio Code si è dimostrato particolarmente efficace, offrendo un supporto completo per il linguaggio e strumenti avanzati per la scrittura e il debug del codice.

1.5 Streamlit

Streamlit rappresenta una risorsa fondamentale nel nostro progetto ed è una libreria open-source che offre numerose funzionalità per la creazione di applicazioni web interattive in Python. Nel contesto del nostro progetto di chatbot, Streamlit si è dimostrato straordinariamente utile per sviluppare l'interfaccia utente, offrendo una visualizzazione chiara e intuitiva delle conversazioni e delle risposte generate dal sistema.

Questo ha consentito agli utenti di interagire direttamente con il chatbot attraverso il proprio browser, senza la necessità di installare alcun software aggiuntivo. Tale accessibilità ha notevolmente ampliato la portata e l'utilità del chatbot, rendendolo facilmente utilizzabile da qualsiasi dispositivo connesso a Internet.

Grazie a queste caratteristiche, siamo riusciti a creare un'interfaccia utente accattivante e altamente funzionale per il nostro chatbot, garantendo un'esperienza di utilizzo ottimale per gli utenti finali.

1.6 Ollama

Per evitare di usare modelli di linguaggi come GPT 3.5- turbo (usato per sviluppare il nostro chatbot) che richiedono l'acquisto di key è possibile utilizzare Ollama.

Fornisce strumenti per sviluppare, addestrare e implementare modelli AI, gestendo l'intero ciclo di vita del modello, dall'addestramento alla distribuzione.

Ollama permette l'accesso e la gestione dei dati necessari per l'addestramento, offrendo anche un'interfaccia utente intuitiva per interagire con i modelli. Si integra facilmente con altre piattaforme e servizi, aiutando le aziende a incorporare funzionalità AI nelle loro applicazioni esistenti. In sintesi, rende più accessibile e gestibile l'uso di modelli AI avanzati per aziende e sviluppatori.

Capitolo 2: Situazione di partenza del progetto

2.1 Fase iniziale

Durante la fase iniziale di progettazione e realizzazione di ParlaMente, il progetto si presentava come un programma Python suddiviso in quattro moduli principali, destinati alla creazione di una piattaforma di chatbot basata su documenti personalizzati. I quattro moduli del programma erano: **bot.py**, **falcon.py** , **main_bot.py** e **chatbot_streamlit_combined.py**.

In particolare, il programma offriva una piattaforma di chatbot che utilizzava documenti caricati dagli utenti come base di conoscenza. Gli utenti potevano caricare documenti tramite l'interfaccia di **bot.py**, che venivano successivamente processati per generare embeddings. Questi embeddings venivano salvati e utilizzati per addestrare un chatbot configurato tramite main_bot.py. Il modulo **chatbot_streamlit_combined.py** gestiva l'interfaccia utente complessiva e le diverse pagine dell'applicazione, mentre **falcon.py** forniva le funzioni di backend per la gestione dei documenti e del modello di linguaggio.

Di seguito vengono illustrate le funzionalità di ciascun modulo, riportando esclusivamente estratti di codice essenziali alla creazione del progetto finale, integrando aspetti teorici e considerazioni sull'utilizzo delle tecnologie coinvolte.

2.1.1 bot.py

Il modulo **bot.py** fornisce un'interfaccia utente tramite **Streamlit** per il caricamento di documenti in formato PDF o TXT, che costituiscono la base di conoscenza del chatbot.

Gli utenti hanno la possibilità di specificare diversi parametri chiave per ottimizzare il funzionamento del chatbot. Tra questi parametri figurano il modello di embeddings, la dimensione dei chunk (ovvero le sezioni in cui vengono suddivisi i documenti) e l'overlap tra questi chunk.

Questa flessibilità consente di adattare il chatbot a diverse esigenze e migliorare l'accuratezza delle risposte generate.

1. **Modello di embeddings**: Gli embeddings sono rappresentazioni vettoriali dei testi che consentono al modello di linguaggio di comprendere meglio il contenuto e il contesto delle informazioni. Gli utenti possono scegliere tra diversi modelli di embeddings disponibili, come quelli forniti da librerie avanzate di Natural Language Processing (NLP) come HuggingFace. La scelta del modello può influenzare significativamente la performance del chatbot, in termini di precisione e pertinenza delle risposte.

2. **Dimensione dei chunk**: La suddivisione dei documenti in chunk più piccoli è un passaggio fondamentale per la generazione degli embeddings. Gli utenti possono determinare la dimensione di questi chunk, bilanciando tra granularità e contesto. Chunk più piccoli possono offrire una maggiore precisione nei dettagli, mentre chunk più grandi possono preservare meglio il contesto.

3. **Overlap tra i chunk**: L'overlap definisce quanto i chunk consecutivi si sovrappongono l'uno con l'altro. Un maggiore overlap può migliorare la continuità e la coerenza delle informazioni tra i chunk, facilitando la comprensione del contesto da parte del modello di linguaggio.

Una volta definiti questi parametri, gli utenti possono scegliere se creare un nuovo archivio di vettori o aggiornare uno esistente. La creazione di un nuovo archivio è utile quando si inizializza il sistema con una nuova base di conoscenza, mentre l'aggiornamento di un archivio esistente consente di aggiungere nuove informazioni mantenendo le conoscenze precedenti.

Dopo aver caricato i documenti tramite l'interfaccia Streamlit, questi vengono processati e suddivisi in chunk in base ai parametri specificati. I chunk vengono poi passati al modulo **falcon.**py**,**che genera gli embeddings. Questo processo di generazione implica l'utilizzo di modelli di deep learning forniti dalla libreria **HuggingFaceEmbeddings**. Questi modelli trasformano i chunk di testo in rappresentazioni numeriche che catturano sia il significato che il contesto dei testi.

Gli embeddings generati vengono quindi salvati in un database o in un archivio di vettori, pronti per essere utilizzati dal chatbot. Questi embeddings costituiscono la base di conoscenza del chatbot, permettendogli di rispondere alle domande degli utenti in modo preciso e contestualmente rilevante.

Gli embeddings sono rappresentazioni numeriche dei documenti, che permettono al modello di linguaggio di comprendere e manipolare i testi in maniera efficiente. Utilizzando tecniche avanzate di Natural Language Processing (NLP), come quelle offerte dalla libreria HuggingFace, il modulo **falcon.py** consente di trasformare i documenti in embeddings. La libreria **HuggingFaceEmbeddings** sfrutta modelli di deep learning per generare queste rappresentazioni, che sono cruciali per il funzionamento del chatbot.

```
# Creazione di un form per l'input del documento
with st.form("document_input"):
    # Caricamento di file PDF e TXT
    document = st.file_uploader(
        "Knowledge Base Documents",
        type=['pdf', 'txt'],
        help=".pdf or .txt file",
        accept_multiple_files=True
    )

    # Creazione di colonne per i campi di input
    row_1 = st.columns([2, 1, 1])

    # Input di testo per il nome del modello di embeddings
    with row_1[0]:
        instruct_embeddings = st.text_input(
            "Model Name of the Instruct Embeddings",
```

```
            value="hkunlp/instructor-xl"
        )

    # Input numerico per la dimensione dei chunk
    with row_1[1]:
        chunk_size = st.number_input(
            "Chunk Size",
            value=200,
            min_value=0,
            step=1
        )

    # Input numerico per l'overlap dei chunk
    with row_1[2]:
        chunk_overlap = st.number_input(
            "Chunk Overlap",
            value=10,
            min_value=0,
            step=1,
            help="Superiore alla dimensione del chunk"
        )

    # Pulsante di invio per il form
    save_button = st.form_submit_button("Save")

# Elaborazione del documento una volta premuto il pulsante di salvataggio
if save_button:
    # Lettura del file caricato
    if document.name[-4:] == ".pdf":
        document = falcon.read_pdf(document)
    elif document.name[-4:] == ".txt":
        document = falcon.read_txt(document)
    else:
        st.error("Verifica che il file caricato sia un .pdf o .txt")

    # Suddivisione del documento in chunk
    split = falcon.split_doc(document, chunk_size, chunk_overlap)
```

```
# Determinazione se creare un nuovo archivio di vettori
create_new_vs = None
if existing_vector_store == "<New>" and new_vs_name != "":
    create_new_vs = True
elif existing_vector_store != "<New>" and new_vs_name != "":
    create_new_vs = False
else:
    st.error(
        """Verifica 'Vector Store to Merge the Knowledge'
        e 'New Vector Store Name'"""
    )

# Generazione degli embeddings e loro memorizzazione
falcon.embedding_storing(
    split,
    create_new_vs,
    existing_vector_store,
    new_vs_name
)
```

2.1.2 chatbot_streamlit_combined.py

Questo modulo ha il compito di gestire la selezione e l'inizializzazione delle diverse pagine dell'applicazione Streamlit, comprese quelle dedicate alla generazione degli embeddings dei documenti e al chatbot basato su RAG (Retrieval-Augmented Generation).

RAG utilizza una fase di retrieval per estrarre informazioni pertinenti da un database o un insieme di documenti, seguita da una fase di generation in cui costruisce una risposta basata sulle informazioni recuperate. Questo processo sfrutta la potenza dei modelli di linguaggio naturale (LLM) per comprendere e rispondere alle richieste degli utenti in modo dettagliato e contestuale.

Il modulo include anche funzioni per la gestione della memoria GPU, assicurando che vi siano risorse sufficienti per l'esecuzione efficiente del modello di linguaggio su GPU. La gestione della memoria è cruciale, soprattutto quando si utilizzano modelli di grandi dimensioni come GPT-3 o GPT-4, che possono richiedere una notevole quantità di risorse computazionali. Ottimizzare l'uso della memoria GPU permette di evitare problemi di out-of-memory (OOM) e garantisce che il modello possa operare alla massima efficienza.

L'interfaccia utente del modulo offre funzionalità per configurare il modello di LLM e le impostazioni del chatbot. Gli utenti possono selezionare diversi parametri, come il modello di embeddings da utilizzare, la dimensione dei chunk di testo e l'overlap tra i chunk. Questi parametri sono fondamentali per ottimizzare la performance del chatbot e per adattarlo alle specifiche esigenze dell'applicazione. Inoltre, il modulo gestisce la sessione di conversazione, permettendo agli utenti di interagire con il chatbot in tempo reale, visualizzare la cronologia delle chat e vedere le fonti dei documenti utilizzati per generare le risposte.

I modelli di Large Language Model (LLM) come GPT-3 e GPT-4 sono al cuore del sistema. Questi modelli utilizzano reti neurali profonde per comprendere e generare linguaggio naturale. Sono addestrati su vasti corpus di testo, il che consente loro di apprendere le strutture grammaticali, i significati delle parole e il contesto in modo dettagliato. Questa conoscenza permette loro di rispondere in maniera coerente e pertinente a una vasta gamma di domande e di generare testi che sono difficilmente distinguibili da quelli scritti da esseri umani.

Successivamente, il progetto è stato ulteriormente raffinato e migliorato, ma la struttura di base e le funzionalità principali descritte qui sono state fondamentali per la sua realizzazione iniziale.

```
# Prepara il modello LLM
if "conversation" not in st.session_state:
    # Se la conversazione non è ancora stata inizializzata, impostala come None
    st.session_state.conversation = None

if token:
```

```
    # Se è stato fornito un token per il modello, prepara il modello RAG-LMM
    st.session_state.conversation = falcon.prepare_rag_llm(
        token, existing_vector_store, temperature, max_length
    )

# Storia della chat
if "history" not in st.session_state:
    # Se non esiste ancora una storia della chat, inizializzala come lista vuota
    st.session_state.history = []

# Documenti di origine
if "source" not in st.session_state:
    # Se non esistono ancora i documenti di origine, inizializzali come lista vuota
    st.session_state.source = []



# Visualizza la chat
for message in st.session_state.history:
    # Per ogni messaggio nella storia della chat
    with st.chat_message(message["role"]):
        # Visualizza il ruolo del mittente (utente o assistente) e il contenuto del
messaggio
        st.markdown(message["content"])

# Fai una domanda
if question := st.chat_input("Fai una domanda"):
    # Aggiungi la domanda dell'utente alla storia della chat
    st.session_state.history.append({"role": "utente", "content": question})
    # Visualizza la domanda dell'utente nella chat
    with st.chat_message("utente"):
        st.markdown(question)

    # Rispondi alla domanda
    answer, doc_source = falcon.generate_answer(question, token)

    # Visualizza la risposta dell'assistente nella chat
```

```
    with st.chat_message("assistente"):
        st.write(answer)
    # Aggiungi la risposta dell'assistente alla storia della chat
    st.session_state.history.append({"role": "assistente", "content": answer})

    # Aggiungi le fonti del documento alla lista dei documenti di origine
    st.session_state.source.append({"domanda": question, "risposta": answer,
"documento": doc_source})
```

2.1.3 falcon.py

Questo modulo costituisce il cuore del sistema, fornendo le funzioni fondamentali per gestire i documenti e il modello di linguaggio. Inoltre, si occupa di coordinare il processo di generazione delle risposte alle domande degli utenti, sfruttando sia il modello di linguaggio che gli embeddings dei documenti.

Il modello di linguaggio impiegato è il "falcon-7b-instruct", mentre per quanto riguarda gli embeddings dei documenti, si fa uso del modello "all-MiniLM-L6-v2". Entrambi questi modelli sono pre-addestrati e sono forniti dalla piattaforma "HuggingFace".

Le funzioni di questo modulo comprendono la gestione dei documenti caricati dagli utenti, l'inizializzazione del modello di linguaggio, la generazione delle risposte alle domande degli utenti e la memorizzazione delle informazioni relative alle interazioni.

In sintesi, questo modulo costituisce il fulcro del sistema, garantendo una gestione efficiente dei documenti e consentendo al sistema di fornire risposte accurate e contestualmente rilevanti agli utenti.

```
# Creazione del chatbot
qa_conversation = ConversationalRetrievalChain.from_llm(
    llm=llm,  # Modello di linguaggio
    chain_type="stuff",  # Tipo di catena (es. stuff)
    retriever=loaded_db.as_retriever(search_kwargs={"k": 3}),  # Recupero di
informazioni
    return_source_documents=True,  # Restituisci i documenti di origine
```

```
    memory=memory,  # Memoria del chatbot
)

return qa_conversation  # Ritorna la conversazione del chatbot

def generate_answer(question, token):
    answer = "Si è verificato un errore"
    if token == "":
        answer = "Inserisci il token di Hugging Face"
        doc_source = ["nessuna fonte"]
    else:
        response = st.session_state.conversation({"question": question})
        answer = response.get("answer").split("Risposta utile:")[-1].strip()
        explanation = response.get("documenti_sorgente", [])
        doc_source = [d.contenuto_pagina for d in explanation]
    return answer, doc_source
```

2.1.4 main_bot.py

Il modulo offre agli utenti la possibilità di personalizzare varie impostazioni del modello di linguaggio LLM. Queste personalizzazioni includono la selezione del modello specifico di LLM da utilizzare, la scelta dell'archivio di vettori, nonché la configurazione di parametri chiave come la temperatura e la lunghezza massima delle risposte generate dal chatbot. Inoltre, il modulo gestisce la sessione di conversazione, mantenendo traccia della cronologia delle chat e delle fonti dei documenti utilizzati per generare le risposte alle domande degli utenti.

```
# Impostazione del LLM
with st.expander("Impostazione del LLM"):
    st.markdown("Questa pagina è utilizzata per conversare con i documenti
caricati")


    # Form per impostare i parametri del LLM
    with st.form("setting"):
        row_1 = st.columns(3)
```

```
        # Input per il token Hugging Face
        with row_1[0]:
            token = st.text_input("Token Hugging Face", type="password")

        # Input per il modello LLM
        with row_1[1]:
            llm_model = st.text_input("Modello LLM", value="tiiuae/falcon-7b-
instruct")

        # Input per gli embeddings di istruzione
        with row_1[2]:
            instruct_embeddings = st.text_input("Embeddings di Istruzione",
value="hkunlp/instructor-xl")

        row_2 = st.columns(3)

        # Selezione dell'archivio di vettori
        with row_2[0]:
            vector_store_list = os.listdir("archivio di vettori/")
            default_choice = vector_store_list.index('naruto_snake') if
'naruto_snake' in vector_store_list else 0
            existing_vector_store = st.selectbox("Archivio di vettori",
vector_store_list, default_choice)

        # Input per la temperatura
        with row_2[1]:
            temperature = st.number_input("Temperatura", value=1.0, step=0.1)

        # Input per la lunghezza massima
        with row_2[2]:
            max_length = st.number_input("Lunghezza massima (in caratteri)",
value=300, step=1)

        # Pulsante per creare il chatbot
        create_chatbot = st.form_submit_button("Crea chatbot")
```

```
# Preparazione del modello LLM
if "conversation" not in st.session_state:
    st.session_state.conversation = None

# Se è presente un token, prepara il modello LLM
if token:
    st.session_state.conversation = falcon.prepare_rag_llm(
        token, existing_vector_store, temperature, max_length
    )
```

Capitolo 3 : ParlaMente

ParlaMente rappresenta un'innovativa soluzione di chatbot interattivo basata su un modello di Retrieval-Augmented Generation (RAG). Il suo obiettivo principale è quello di fornire risposte contestuali e accurate a domande basate su documenti PDF uniti e indicizzati, concentrati sui registri della Camera dei Deputati.

Questo progetto sfrutta tecnologie avanzate di Natural Language Processing (NLP) per elaborare e comprendere il contenuto di tali documenti, consentendo al chatbot di generare risposte pertinenti e informati. L'approccio RAG combina tecniche di recupero delle informazioni con la generazione di risposte, garantendo una maggiore pertinenza e contestualità nelle risposte fornite.

Per lo sviluppo di ParlaMente sono stati creati quattro script Python che verranno esaminati in dettaglio in seguito. Questi script formano il cuore del chatbot, coordinando il caricamento, il preprocessamento e l'elaborazione dei documenti PDF, nonché la gestione delle interazioni con gli utenti e la generazione delle risposte.

È importante sottolineare che ParlaMente è stato utilizzando le API di OPENAI per la generazione delle risposte, ma si fornisce anche un file di utility ulteriore totalmente open source. Questa differenziazione consente di esplorare e valutare diverse tecnologie e approcci nell'implementazione di un chatbot basato su RAG.

3.1 Parlamente API OPENAI

Nel nostro chatbot, abbiamo suddiviso la logica di funzionamento in diversi script per una gestione modulare e organizzata. Tali script includono **DataAnalysis.py**, **merger.py**, **OpenAI_utility.py**, e **chatbot_parlamente.py , coffee_theme.css , custom_theme.css** . Ogni modulo svolge un ruolo specifico nel funzionamento complessivo del nostro chatbot. Analizziamo ora ogni script nel dettaglio.

3.1.1 DataAnalysis.py

Nel contesto dell'analisi testuale dei documenti PDF, la suddivisione e il conteggio delle parole sono procedure fondamentali per estrarre informazioni rilevanti. In questo contesto, esploreremo una pratica implementazione di tali operazioni utilizzando il linguaggio di programmazione Python. Per svolgere questa analisi, ci avvaliamo delle seguenti librerie:

- *pypdf*: una libreria Python per manipolare file PDF, che ci consente di estrarre il testo dai documenti PDF per utilizzarla è necessario eseguire da linea di comando: *pip install pypdf2*
- *math*: una libreria standard di Python per le operazioni matematiche, necessaria per calcolare la deviazione standard dei conteggi delle parole.

Il codice Python definisce una funzione denominata *split_and_count_tokens* che prende come input il percorso di un documento PDF e una parola su cui dividere il testo. La funzione suddivide il testo del PDF in sezioni in base alla parola specificata e conta il numero di parole in ciascuna sezione.

La funzione utilizza due funzioni ausiliarie:

1. *count_tokens*: questa funzione conta il numero di token (parole) in un testo fornito.
2. *split_text_recursive*: questa funzione divide il testo in sezioni in base alla parola specificata, ricorsivamente, e restituisce una lista di sezioni.

Il testo viene estratto dal documento PDF utilizzando la libreria *pypdf* e viene suddiviso in sezioni utilizzando la parola specificata. Successivamente, vengono conteggiati i token in ciascuna sezione.

Utilizzando Python e le librerie appropriate, come *pypdf* e *math*, è possibile automatizzare il processo di estrazione di informazioni significative da documenti , rendendo l'analisi più efficiente e accurata.

```
from pypdf import PdfReader
import math

def split_and_count_tokens(pdf_path, split_word):
  #function to split the text in a PDF by a Blyword and count tokens in each part."
    def count_tokens (text):
        #"Count the number of tokens in a given text.
        return len(text.split())

    def split_text_recursive(text, split_word):
   #Recursively split the text on a split_word, returning a flat list of sections."
        parts = []
        if split_word in text:
            splitted_text = text.split(split_word)
            for subtext in splitted_text:
                parts.extend(split_text_recursive(subtext, split_word))

 else:
            parts.append(text)
        return parts

    with open(pdf_path, 'rb') as file:
        pdf = PdfReader(file)
        num_pages = len(pdf .pages) # Use the len() function to get the number of
pages
        text = ''
        for page_num in range(num_pages):
            page = pdf.pages[page_num] # Correct reference to the page object
            extracted_text=page.extract_text()
            if extracted_text: # Check if text was extracted
                text += extracted_text
```

```
    splitted_texts = split_text_recursive(text, split_word)
    token_counts = [count_tokens(subtext) for subtext in splitted_texts if
subtext.strip()]
    return token_counts

# Usage example
pdf_path = "merged_RegistroCmeraDeputati.pdf"
split_word = "-"
token_counts = split_and_count_tokens(pdf_path, split_word)
mean_token_counts = sum(token_counts) / len(token_counts)
var_token_counts = sum((x - mean_token_counts) ** 2 for x in token_counts) /
len(token_counts)
print(f"Mean Token Counts: {mean_token_counts:.2f} word per section")
numb_of_words = mean_token_counts*4
print(f"Total number of tokens: {numb_of_words}")
std = math.sqrt(var_token_counts)
print(f"std of Token Counts: {std:.2f} tokens^2 per section")
```

3.1.2 merger.py

Per eseguire questo codice, è necessario installare la libreria *PyPDF*. Puoi farlo tramite pip eseguendo il seguente comando : *pip install PyPDF*.

La libreria *PyPDF* di Python offre una serie di funzionalità per manipolare i file PDF. Il codice fornito presenta una funzione denominata *merge_pdfs*, la quale prende come input il percorso di una cartella contenente file PDF e un percorso di output dove salvare il PDF unito. Utilizzando la classe *PdfMerger* fornita dalla libreria *PyPDF*, questa funzione attraversa in modo ricorsivo tutti i file nella cartella specificata e li unisce in un unico file PDF, rispettando l'ordine di lettura all'interno della cartella.

Questo codice funziona su file PDF presenti all'interno della cartella specificata dalla variabile *folder_path* e li unisce tutti in un unico file PDF salvato al percorso specificato dalla variabile *output_path*.

Un elemento cruciale all'interno di questo codice è l'importazione della libreria *os*, che fornisce funzionalità per interagire con il sistema operativo sottostante. Questa libreria è essenziale per operazioni come la navigazione tra le directory *(os.walk)* e la gestione dei percorsi dei file *(os.path.join).* Senza di essa, sarebbe difficile, se non impossibile, scorrere i file all'interno di una directory e ottenere i percorsi completi dei singoli file.

```
import os
from pypdf import PdfMerger
#Questa funzione merge_pdfs prende come input il percorso della cartella che
contiene i PDF (folder_path) e il percorso dove salvare il PDF unito (output_path).
#La funzione utilizza os.walk per attraversare tutti i file nella cartella e unisce
solo quelli che terminano con .pdf (ignorando maiuscole/minuscole).
def merge_pdfs(folder_path, output_path):
    merger = PdfMerger()

    for root, dirs, files in os.walk(folder_path):
        for file in files:
            if file.lower().endswith('.pdf'):
                file_path = os.path.join(root, file)
                merger.append(file_path)

    merger.write(output_path)
    merger.close()
# Utilizzo:
folder_path = 'RegistroCameraDeputatiPorgetto'  # Cartella con i PDF all'interno
del progetto
output_path = 'merged_RegistroCmeraDeputati.pdf'  # Nome del file unito all'interno
del progetto
merge_pdfs(folder_path, output_path)
```

3.1.3 OpenAI_utility.py

L'obiettivo è quello di sviluppare un sistema conversazionale in grado di rispondere in modo intelligente alle domande degli utenti, combinando retrieval di informazioni e generazione di risposte e per questo sono state usate numerose librerie Python. Le librerie utilizzate sono :

- *os*: una libreria standard di Python per operare con il sistema operativo, utile per gestire variabili d'ambiente e percorsi dei file.
- *openai*: Una libreria per l'accesso alle API di OpenAI, che ci consente di utilizzare modelli di linguaggio avanzati per generare risposte ai messaggi degli utenti, nel nostro caso usiamo come linguaggio GPT-3.5 turbo . Per utilizzarla è stata installata tramite : *pip install openai.*
- *langchain_openai*: una libreria che offre funzionalità aggiuntive per l'integrazione di modelli OpenAI nel flusso di lavoro di NLP, installata mediante *pip install langchain_openai.*
- *streamlit*: una libreria Python per creare applicazioni web interattive e data-driven in modo rapido e intuitivo. Tale libreria è stata installata per l'utlizzo con il comando :*pip install streamlit.*
- *pypdf*: una libreria per la manipolazione dei file PDF, che ci permette di estrarre il testo da documenti PDF che deve essere installata mediante *pip install pypdf2.*

- *langchain*: una libreria che fornisce strumenti per l'elaborazione del testo, come lo splitter di testo ricorsivo e la gestione della memoria durante le conversazioni. Per usufruirne è necessario eseguire : *pip install langchain.*
- *langchain_community*: una raccolta di moduli aggiuntivi per la libreria *langchain*, inclusa la gestione dei vettori e la creazione di catene di elaborazione conversazionale, anch'essa deve essere installata separatamente con *pip install langchain_community*.
- *faiss*: una libreria per la ricerca efficiente e il clustering di grandi collezioni di vettori, utilizzata per eseguire ricerche di similarità su grandi dataset di embedding. L'utlizzo può avvenire solo dopo aver effettuato *pip install faiss.*

Il codice implementa diverse funzionalità per la creazione di un chatbot basato su retrieval di informazioni e generazione di risposte. Le funzioni create sono:

1. *merge_pdfs*: una funzione che unisce file PDF presenti in una cartella specifica in un unico documento PDF.
2. *read_pdf*: una funzione che legge il testo da un documento PDF utilizzando la libreria *pypdf*.
3. *split_doc*: una funzione che suddivide documenti di grandi dimensioni in chunk più piccoli per consentire il processamento da parte del modello.
4. *embedding_storing*: una funzione che genera gli embedding del documento e li memorizza in un vector store per future ricerche di similarità.
5. *prepare_rag_llm*: una funzione che inizializza e prepara il modello conversazionale RAG (Retrieval-Augmented Generation) con una configurazione specifica.
6. *generate_answer*: una funzione che genera risposte alle query dell'utente utilizzando il modello conversazionale RAG configurato in precedenza.

```
import os
import openai
```

```
from langchain_openai import OpenAIEmbeddings
from langchain_openai import ChatOpenAI
#Streamlit è una libreria Python per creare rapidamente applicazioni web
interattive e data-driven.
import streamlit as st
#PdfReader è una classe della libreria  pypdf che consente di leggere e manipolare
file PDF
from pypdf import PdfReader
from pypdf import PdfMerger
#RecursiveCharacterTextSplitter è uno strumento per suddividere testi lunghi in
segmenti più piccoli in modo ricorsivo, mantenendo la coerenza delle frasi. È utile
per pre-processare testi per analisi successive.
from langchain.text_splitter import RecursiveCharacterTextSplitter
#FAISS (Facebook AI Similarity Search) è una libreria per la ricerca efficiente e
il clustering di grandi collezioni di vettori. Viene utilizzata per eseguire
ricerche di similarità su grandi dataset di embedding.
from langchain_community.vectorstores import FAISS
#ConversationalRetrievalChain è una catena di elaborazione che combina retrieval di
informazioni e generazione di risposte in un contesto conversazionale, utile per
chatbot e assistenti virtuali.
from langchain.chains import ConversationalRetrievalChain
#ConversationBufferWindowMemory è una struttura di memoria che memorizza le ultime
N interazioni di una conversazione. È utile per mantenere il contesto durante una
conversazione lunga, migliorando la coerenza delle risposte.
from langchain.memory import ConversationBufferWindowMemory

os.environ["OPENAI_API_KEY"] = "sk-proj-
LpHTq3H3G5pP1Zyko6DcT3BlbkFJWOvn5VfBPTdM0CvVFSTX"

#Per ottenere la api key dalla var. d'ambiente
openai.api_key =os.getenv("OPENAI_API_KEY")


#Questa funzione merge_pdfs prende come input il percorso della cartella che
contiene i PDF (folder_path)
#e il percorso dove salvare il PDF unito (output_path). La funzione utilizza
os.walk per attraversare
```

```
#tutti i file nella cartella e unisce solo quelli che terminano con .pdf (ignorando
maiuscole/minuscole).
def merge_pdfs(folder_path, output_path):
    merger = PdfMerger()

    for root, dirs, files in os.walk(folder_path):
        for file in files:
            if file.lower().endswith('.pdf'):
                file_path = os.path.join(root, file)
                merger.append(file_path)

    merger.write(output_path)
    merger.close()


#funzione che consente di leggere i pdf e ritornarne il contenuto testuale tramite
la libreria PdfReader;
#sostanzialmente legge le pagine del pdf attraverso un ciclo: per ogni pagina
all'interno
#dell'oggetto reader, generato istanziando la funzione PdfReader a cui viene
passato il file ovvero il suo
#specifico path, permette di estrarre il testo e di ritornare il documento
concatenando (+=) il testo
#di tutto il pdf. Quindi il risultato finale di questa funzione è che ritorna il
testo del pdf
#concatenando tutti le stringhe appartenenti ad ogni pagina del pdf;
#Inoltre sostituisco il next-line + "-" con #, in modo da realizzare sono uno
splitting in chunks più
#consistente del documento;
def read_pdf(file):
    document = ""

    reader = PdfReader(file)
    for page in reader.pages:
        document += page.extract_text()
        document = document.replace("\n-", "\n#")
```

```
    return document



#Splittiamo i documenti di grande dimensioni in chunks, poichè il modello non può gestire input più grandi della sua context Length che nella maggior parte dei casi è 2048 token!
#Ovviamente vogliamo realizzare uno split del documento che favorisca il recupero corretto delle informazioni.
#Generiamo chunks fino a 200 token(o caratteri) attraverso separatori come double new line o space applicati sul documento;
#Una volta individuato un chunk, realizziamo un secondo chunk non partendo dalla fine di quest'ultimo, ma tornando indietro di tot
#caratteri al fine non rischiare di perdere informazioni con lo split; cioè da un lato può portare a rindondanza, ma assicuro maggiormente di preservare il significato semanatico nei chunk prodotti dal documento.

#funzione per realizzare il recursive character text splitter;
#In tale funzione vengono passati i parametri settati a priori chunk size e chunk overlapping ottimali per i testi forniti;
#viene istanziato l'oggetto "splitter" a partire dalla classe RecursiveCharacterTextSplitter in cui vengono passati i parametri precedenti;
#su splitter viene richiamata la funzione split_text con cui viene eseguito uno split del documento ritornato da read_text sulla base del conteggio dei caratteri; viene dunque ritornato il documento splittato;
#su splitter viene richiamata la funzione create_documents che crea documenti, a partire dal testo splittato creato prima, in un formato per il processing del documento;
#al termine di tutto, la funzione dunque ritorna i chunk del documento originale;
def split_doc(document, chunk_size, chunk_overlap):

    splitter = RecursiveCharacterTextSplitter( ##aggiungi parte Split dell'nb!
        separators="#",
        chunk_size=chunk_size,
        chunk_overlap=chunk_overlap
    )
    split = splitter.split_text(document)
```

```
    split = splitter.create_documents(split)

    return split


#codice per vedere i chunk generati
#result = split_doc(read_pdf("merged_RegistroCmeraDeputati.pdf"), 330, 50)
#print(result[0])
#print(result[1])
#print(result[2])



#una volta creati i chunks, realizziamo l'embedding del documento e il salvataggio
in un vector store per un futuro retrieval;
#tale funzione prendere in ingresso i chunks generati con split_doc e dei parametri
di controllo nel caso in cui si sta creando un nuovo vector store, con un nuovo
nome, o se se ne sta ulitizzando creato in precedenza;
def embedding_storing( split, create_new_vs, existing_vector_store, new_vs_name):


        #Viene usato un pretrained sentence transformer model per generare
l'embedding;
        #In tal caso si sfruttano i modelli di Embedding offerti da OpenAI (senza
specificare il modello di default si utilizza text-embedding-ada-002)

        instructor_embeddings = OpenAIEmbeddings()

        #Viene generato un vs tramite la classe FAISS per realizzare la similarity
search,
        #in particolare, applicando il metodo from_documents che a partire dai
chunks dal modello di embedding precedente
        #ho dunque generato il database;
        #new db = FAISS.from_documents(split, instructor_embeddings)
        db = FAISS.from_documents(split, instructor_embeddings)
```

```
        #Se è stato creato un nuovo vs, lo salvo in una CACHE definita a partire dal nome del nuovo vs;
        #ciò mi permetterà successivamente, dopo aver istanziato almeno la prima volta il vs, di poterlo richiamare subito
        #senza dover generare di nuovo gli embedding ed inserirli nel db;
        #quindi, nel caso il vs esiste già, viene recuperato dalla cache, si ha il merge dei due db e risalvato in cache;
        if create_new_vs == True:
            # Save db
            db.save_local("vector store/" + new_vs_name)
        else:
            # Load existing db
            load_db = FAISS.load_local(
                "vector store/" + existing_vector_store,
                instructor_embeddings,
                allow_dangerous_deserialization=True
            )
            # Merge two DBs and save
            load_db.merge_from(db)
            load_db.save_local("vector store/" + new_vs_name)



#funzione per inizializzare e preparare il RAG conversational model con una data configurazione
def prepare_rag_llm(vector_store_list, temperature, max_length):


    #istanzio nuovamente il modello di embedding precedente;
    instructor_embeddings = OpenAIEmbeddings()

    # carico il mio db attraverso load_local della classe FAISS passando il file path del vector_store su cui lavorerà il RAG,
    #il modello di embedding, ed una serie di parametri (vedi a cosa serve allow_dangerous_deserialization)
    loaded_db = FAISS.load_local(
```

```
        f"vector store/{vector_store_list}", instructor_embeddings,
allow_dangerous_deserialization=True
    )

    #istanzio l'LLM richiamando il modello di GPT-3.5-turbo attraverso la classe
per l'integrazione
    #di modelli OpenaAI, settando la temperature e la massima lungezza della
risposta che viene data in output;
    #L'API_KEY viene ottenuta direttamente a partire dalle variabili d'ambiente;
    #NOTA: TEMPERATURA = parametro di entropia che va da 0 a 1 e che determina
quanto la risposta del chatbot sarà creativa (andando verso l'1), ovvero una
risposta più o meno deterministica.
    llm = ChatOpenAI(
        model="gpt-3.5-turbo", #Il modello GPT-3.5-turbo può gestire fino a 4096
token per richiesta. Questo limite include sia i token di input che quelli di
output generati dal modello. Ad esempio, se invii 2000 token come input, il modello
può generare fino a 2096 token in output.
        temperature= temperature,
        max_tokens = max_length
    )

    #setto ConversationBuffer che consente al chat bot di recuperare informazioni a
partire dalla storia
    #della chat
    memory = ConversationBufferWindowMemory(
        k=2, #recupera info a partire dalle ultime 2 interazioni;
        memory_key="chat_history", #tali info verrare usate in streamlit
        output_key="answer", #tali info verrare usate in streamlit
        return_messages=True,
    )

    # Create the chatbot:
    #setto la question-answer chain attraverso la chain
ConversationalRetrievalChain
    #fornita dal LangChain, ovvero un metodo che recupera l'LLM definito prima, il
db
```

```
    #settato in modalità as_retriever, ovvero pronto a cercare i documenti
necessari,
    #e settando il parametro di memoria precedente;
    qa_conversation = ConversationalRetrievalChain.from_llm(
        llm=llm,
        chain_type="stuff", #refine #il parametro chain_type="stuff" indica che i
tre documenti più attinenti recuperati verranno combinati insieme e passati al
modello di linguaggio in una singola richiesta per generare una risposta alla query
        retriever=loaded_db.as_retriever(search_kwargs={"k": 3}), #Sfrutto i top 3
documenti più attinenti alla query
        return_source_documents=True, #serve a indicare se i documenti sorgente
utilizzati per generare la risposta devono essere restituiti insieme alla risposta
stessa.
        memory=memory,
    )

    #alla fine viene ritornata la chain finale di question_answer
    return qa_conversation

#funzione per generare risposte alle query dell'utente usando un il conversational
model realizzato prima:
#prima c'era token
def generate_answer(question):
    answer = "Si è verificato un errore" #Risposta di Default in caso di errore
    response = st.session_state.conversation({"question": question}) #Viene
processata la domanda attraverso il conversational model
    answer = response.get("answer").split("Helpful Answer:")[-1].strip() #Viene
estratta la risposta da response
    explanation = response.get("source_documents", [])
    doc_source = [d.page_content for d in explanation] #Vengono collezionati i
documenti che hanno contribuito alla risposta;

    return answer, doc_source
```

3.1.4 chatbot _parlamente.py

L'obiettivo è sviluppare un sistema conversazionale che risponda in modo intelligente alle domande degli utenti, combinando il retrieval di informazioni e la generazione di risposte. Per raggiungere questo obiettivo, utilizziamo diverse librerie Python, tra quelle ancora non utilizzate troviamo:

- *torch*: una libreria per il calcolo scientifico e il machine learning. Viene utilizzata per la gestione della memoria GPU, essenziale per i modelli di deep learning ed installata mediante: *pip install torch*.
- *gc*: libreria per la gestione della garbage collection in Python, utile per liberare memoria non più utilizzata.
- *time*: libreria standard di Python per la gestione del tempo, utilizzata per inserire pause durante l'esecuzione del codice.
- *pynvml*: Libreria per interfacciarsi con la NVIDIA Management Library (NVML), utilizzata per monitorare e gestire la memoria GPU. L'installazione è avvenuta tramite : *pip install nvidia-ml-py3*.
- *OpenAI_utility*: il modulo personalizzato descritto precedentemente contenente funzioni specifiche per gestire i modelli di linguaggio di OpenAI e le operazioni di embedding dei documenti.
- *streamlit.components.v1*: modulo di Streamlit per creare componenti web personalizzati, utilizzato per migliorare l'interfaccia utente.
- *streamlit.delta_generator*: utilizzata per aggiornare dinamicamente l'interfaccia utente.

Il codice implementa diverse funzionalità quali:

1. *clear_gpu_memory* : svuota la cache della memoria GPU e forza la garbage collection per liberare memoria non più utilizzata.
2. *wait_until_enough_gpu_memory* : verifica se è disponibile una quantità minima di memoria GPU libera, effettuando tentativi ripetuti con un intervallo di attesa configurabile. Se la memoria richiesta non è disponibile dopo il numero massimo di tentativi, solleva un'eccezione.
3. *main*: funzione principale che gestisce la sequenza di esecuzione dell'applicazione. Chiama le funzioni di gestione della memoria e avvia le pagine di embedding dei documenti e del chatbot.
4. *load_css*: carica file CSS specifici e personalizza l'aspetto dell'applicazione.
5. *display_chatbot_page*: configura e visualizza la pagina del chatbot. Carica i temi CSS, gestisce la sidebar per le informazioni sulla sessione e lo storico delle chat, e implementa un bottone per cambiare tema. Prepara il modello di linguaggio e gestisce l'interazione con l'utente, visualizzando le risposte del chatbot.
6. *display_message*: visualizza i messaggi nella chat dell'applicazione, differenziando tra messaggi dell'utente e dell'assistente.
7. *display_document_embedding_page*: verifica se un vector store esiste e, se non esiste, legge un PDF, lo suddivide in chunk, e crea un vector store utilizzando OpenAI_utility.

```
import streamlit as st
import os
import OpenAI_utility
import torch
import time
import gc
```

```
import streamlit.components.v1 as components
from pynvml import nvmlInit, nvmlDeviceGetHandleByIndex, nvmlDeviceGetMemoryInfo
from streamlit.delta_generator import DeltaGenerator  # Import DeltaGenerator for
updating messages

os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"

#progress_bar
#main_placeholder = st.empty()
#def main_place(message="The Task Is Finished !!!!"):
  # main_placeholder.text(message)

# Memory management functions
def clear_gpu_memory():
    torch.cuda.empty_cache()
    gc.collect()

#utility per la gestione della memoria
def wait_until_enough_gpu_memory(min_memory_available, max_retries=10,
sleep_time=5):
    nvmlInit()
    handle = nvmlDeviceGetHandleByIndex(torch.cuda.current_device())

    for _ in range(max_retries):
        info = nvmlDeviceGetMemoryInfo(handle)
        if info.free >= min_memory_available:
            break
        print(f"Waiting for {min_memory_available} bytes of free GPU memory.
Retrying in {sleep_time} seconds...")
        time.sleep(sleep_time)
    else:
        raise RuntimeError(f"Failed to acquire {min_memory_available} bytes of free
GPU memory after {max_retries} retries.")

def main():
    # Call memory management functions before starting Streamlit app
    #min_memory_available = 1 * 1024 * 1024 * 1024  # 1GB
```

```
    clear_gpu_memory()
    #wait_until_enough_gpu_memory(min_memory_available)

    #NOTA: l'ordine delle chiamate nel main ci garantisce che
display_document_embedding_page()
    #venga eseguita e completata prima che display_chatbot_page() venga chiamata.
    #Siamo dunque certi che il vector sia gia presente o che venga creato prima di
interagire col chatbot
    display_document_embedding_page()

    #funzione per la visualizzazione del chatbot, da chiamare SOLO dopo la
creazione del vectore store
    display_chatbot_page()

def load_css(file_name): #funzione per caricare il tema .css adatto
    with open(file_name) as f:
        st.markdown(f"<style>{f.read()}</style>", unsafe_allow_html=True)

def display_chatbot_page():

    #per configurare logo e titolo della pagina nel browser
    st.set_page_config(
        page_title="ParlaMente",

page_icon="https://raw.githubusercontent.com/alessiamanna/text_mining/main/immagine
.png"
    )

    #sono stati creati due temi, coffee theme con una palette di colori caldi, e
custom theme con una palette di colori più classica
    coffee_theme_css = 'coffee_theme.css'
    custom_theme_css = 'custom_theme.css'

    #caricamento del tema di default
    if 'theme' not in st.session_state:
        st.session_state.theme = coffee_theme_css
```

```
    load_css(st.session_state.theme)

    #aggiungi sidebar con informazioni sulla sessione e sullo storico delle chat
    with st.sidebar:
        container = st.container()
        with container:
            st.write("<style> .lower-text { margin-top: 10px; } </style>",
unsafe_allow_html=True)
            st.write('<div class="lower-text">Storico delle chat e info sulla
sessione</div>', unsafe_allow_html=True)
            st.write(st.session_state.get('source', 'No data available'))


    #link di riferimento per le icone
    bot =
"https://raw.githubusercontent.com/alessiamanna/text_mining/main/immagine.png"
    user =
"https://raw.githubusercontent.com/alessiamanna/text_mining/main/user.png"
    doc_logo =
"https://raw.githubusercontent.com/alessiamanna/text_mining/main/doc.png"


    #url da sostituire con collegamento a github per la documentazione
    url = 'https://stackoverflow.com'

    #container per gestire più agevolmente i tasti per la documentazione e per lo
switch dei temi
    container_2 = st.container()
    with container_2:
        buttonDoc = f'''
        <a href="{url}">
            <button class = "customBtn">
                <img src="{doc_logo}" alt = "logo" style="width:50px; position:
relative;">
            </button>
        </a>
        '''
```

```
        st.markdown(buttonDoc, unsafe_allow_html=True)
        # Add a button to switch themes
        if st.button('Switch Theme'):
            if st.session_state.theme == coffee_theme_css:
                st.session_state.theme = custom_theme_css
            else:
                st.session_state.theme = coffee_theme_css
            # Reload CSS after theme change
            load_css(st.session_state.theme)

    #titolo della pagina
    st.markdown(f"""<h1 style='text-align:center;' className='stTitle'> ParlaMente
<img src="{bot}" alt="logo" style="width:65px; position: relative; bottom: 5px;">
    </h1>
    """, unsafe_allow_html=True)

    #sottotitolo della pagina
    st.markdown("""
    <div style='text-align: center; margin-bottom: 30px;'>
        <p class="customHdr" style='font-family: "Roboto", sans-serif; font-size:
1.2em;'>
           Il Tuo Deputato Digitale, Sempre a Disposizione!
        </p>
    </div>
    """, unsafe_allow_html=True)


    # Prepare the LLM model
    if "conversation" not in st.session_state:
        st.session_state.conversation = None

    VECTOR_STORE = "first_CameraDep"
    TEMPERATURE = 0.5
    MAX_LENGTH = 300

    st.session_state.conversation = OpenAI_utility.prepare_rag_llm(
        VECTOR_STORE, TEMPERATURE, MAX_LENGTH
```

```
    )

    # Chat history
    if "history" not in st.session_state:
        st.session_state.history = []

    # Source documents
    if "source" not in st.session_state:
        st.session_state.source = []

    # Display chats
    for message in st.session_state.history:
        display_message(message["role"], message["content"], user if
message["role"] == "user" else bot)

    # Ask a question
    if question := st.chat_input("Fammi una domanda"):
        # Append user question to history
        st.session_state.history.append({"role": "user", "content": question})
        # Add user question
        #with st.chat_message("user", avatar = user):
            #st.markdown(question)

        display_message("user", question, user) #funzione per mostrare i messaggi
        # Answer the question
        answer, doc_source = OpenAI_utility.generate_answer(question)
        #risposta intera
        #with st.chat_message("assistant"):
            #st.write(answer)

        # Display the answer one word at a time
        words = answer.split()
        message_placeholder = st.empty()
        partial_answer = ""
        for i in range(len(words)):
            partial_answer += words[i] + " "
            message_placeholder.markdown(f"""
```

```
                <div class="assistant-message">
                    <img src="{bot}" class="avatar">
                    <div class="chat-bubble assistant-bubble">
                        {partial_answer.strip()}
                    </div>
                </div>
            """, unsafe_allow_html=True)
            time.sleep(0.06)  # Adjust the sleep time to control the speed of word
display
        # Append assistant answer to history
        st.session_state.history.append({"role": "assistant", "content": answer})

        # Append the document sources
        st.session_state.source.append({"question": question, "answer": answer,
"document": doc_source})

def display_message(role, content, avatar_url):
    message_class = "user-message" if role == "user" else "assistant-message"
    bubble_class = "user-bubble" if role == "user" else "assistant-bubble"
    if role == "user":
        st.markdown(f"""
            <div class="{message_class}">
                <div class="chat-bubble {bubble_class}">
                    {content}
                </div>
                <img src="{avatar_url}" class="avatar">
            </div>
        """, unsafe_allow_html=True)
    else:
        st.markdown(f"""
            <div class="{message_class}">
                <img src="{avatar_url}" class="avatar">
                <div class="stChatMessage {bubble_class}">
                    {content}
                </div>
            </div>
        """, unsafe_allow_html=True)
```

```
def display_document_embedding_page():

    #se il vector_store nominato first_CameraDep esiste gia, non lo ricreo, se
invece non esiste allora lo creo con lo stesso nome
    if not os.path.exists("vector store/first_CameraDep"):

        print("assente")

        #richiamo read_pdf
        combined_content =
OpenAI_utility.read_pdf("merged_RegistroCmeraDeputati.pdf")
        #splitto il documento con chunk-size=330 e overlapping=50
        split = OpenAI_utility.split_doc(combined_content, 330, 50)
        #creo il vector_store
        OpenAI_utility.embedding_storing(split, True, "first_CameraDep",
"first_CameraDep")


if __name__ == "__main__":
    main()
```

3.1.5 Interfaccia

Per personalizzare l'interfaccia del nostro chatbot Parlamente, abbiamo creato due fogli di stile: **coffee_theme.css , custom_theme.css**. La necessità di due fogli di stile distinti nasce dall'integrazione di una funzionalità che permette all'utente di cambiare il tema dell'interfaccia tramite un apposito pulsante. Seguono i codici commentati dettagliatamente.

coffee_theme.css

```
/* Importa il font 'Roboto' da Google Fonts */
@import
url('https://fonts.googleapis.com/css2?family=Roboto:wght@300;400;700&display=swap'
);
```

```css
/* Stile principale dell'app Streamlit */
.stApp {
    background-color: #ede8d4; /* Colore di sfondo della pagina */
    color: #000000; /* Colore del testo */
}

/* Stile per i bottoni di Streamlit */
.stButton>button {
    background-color: #5f3c07; /* Colore di sfondo del bottone */
    color: white; /* Colore del testo del bottone */
    position: absolute; /* Posizionamento assoluto */
    right: 50px; /* Posizione a destra */
    top: -40px; /* Posizione in alto */
}

/* Stile per il titolo principale */
.stTitle {
    font-family: 'Roboto', sans-serif; /* Font 'Roboto' */
    font-size: 2.5em; /* Dimensione del testo */
    color: #5e3a05; /* Colore del testo */
    text-align: center; /* Allineamento centrale */
    margin-bottom: 0; /* Margine inferiore zero */
}

/* Stile per gli elementi con classe st-ae */
.st-ae {
    background-color: #dcd6c0; /* Colore di sfondo */
    color: #000000; /* Colore del testo */
    border-color: #5e3a05; /* Colore del bordo */
}

/* Stile per gli elementi attivi con classe st-ae */
.st-ae:active {
    border-color: orange; /* Colore del bordo quando attivo */
}
```

```css
/* Stile per l'header */
.st-emotion-cache-12fmjuu {
    background-color: #dcd6c0; /* Colore di sfondo dell'header */
}

/* Stile per una specifica sezione */
.st-emotion-cache-uhkwx6 {
    background-color: #ede8d4; /* Colore di sfondo */
}

/* Stile per i bottoni personalizzati */
.customBtn {
    background-color: #dcd6c0; /* Colore di sfondo */
    border-radius: 30%; /* Bordi arrotondati */
}

/* Stile per l'header */
.stHeader {
    background-color: #5e3a05; /* Colore di sfondo */
    font-family: 'Roboto', sans-serif; /* Font 'Roboto' */
    font-size: 1.5em; /* Dimensione del testo */
    color: #085457; /* Colore del testo */
}

/* Stile per gli espandibili */
.stExpander {
    font-family: 'Roboto', sans-serif; /* Font 'Roboto' */
    font-size: 1em; /* Dimensione del testo */
    color: #5e3a05; /* Colore del testo */
}

/* Stile per la sidebar */
.stSideBar {
    font-family: 'Roboto', sans-serif; /* Font 'Roboto' */
    font-size: 1em; /* Dimensione del testo */
    color: #5e3a05; /* Colore del testo */
    background-color: #dcd6c0; /* Colore di sfondo */
```

```
}

/* Stile per il markdown */
.stMarkdown {
    font-family: 'Roboto', sans-serif; /* Font 'Roboto' */
    color: #5e3a05; /* Colore del testo */
}

/* Stile per chiavi e valori degli oggetti */
.object-key-val {
    background-color: #dcd6c0; /* Colore di sfondo */
}

/* Stile per i titoli personalizzati */
.customHdr {
    color: #5e3a05; /* Colore del testo */
}

/* Posizionamento degli elementi a 20px dal fondo */
.st-emotion-cache-7ym5gk {
    bottom: 20px;
}

/* Stile per una specifica sezione */
.st-emotion-cache-6qob1r {
    background-color: #dcd6c0; /* Colore di sfondo */
}

/* Stile per i messaggi degli utenti */
.user-message {
    display: flex; /* Layout flex */
    justify-content: flex-end; /* Allineamento a destra */
    align-items: center; /* Allineamento centrale */
    margin-bottom: 10px; /* Margine inferiore */
}

/* Stile per i messaggi dell'assistente */
```

```css
.assistant-message {
    display: flex; /* Layout flex */
    justify-content: flex-start; /* Allineamento a sinistra */
    align-items: center; /* Allineamento centrale */
    margin-bottom: 10px; /* Margine inferiore */
    padding: 10px; /* Padding */
}

/* Stile per i messaggi della chat */
.stChatMessage {
    max-width: 70%; /* Larghezza massima */
    padding: 10px; /* Padding */
    border-radius: 10px; /* Bordi arrotondati */
    display: inline-block; /* Display inline-block */
    font-size: 16px; /* Dimensione del testo */
}

/* Stile per l'icona di invio */
.eyeqlp51 {
    position: absolute; /* Posizionamento assoluto */
    top: 10px; /* Posizione in alto */
    left: -12px; /* Posizione a sinistra */
}

/* Stile per elementi in hover */
.st-emotion-cache-1hkc054:hover {
    background-color: transparent; /* Colore di sfondo trasparente */
    color: rgb(255,255,255); /* Colore del testo bianco */
}

/* Stile per le bolle dei messaggi degli utenti */
.user-bubble {
    background-color: #cfb179; /* Colore di sfondo */
    padding: 8px; /* Padding */
    border-radius: 10px 10px 0 10px; /* Bordi arrotondati */
    text-align: right; /* Allineamento a destra */
    color: #302e2b; /* Colore del testo */
```

```
    max-width: 70%; /* Larghezza massima */
    overflow-wrap: break-word; /* Gestione del testo a capo */
}

/* Stile per le bolle dei messaggi dell'assistente */
.assistant-bubble {
    background-color: #faf8e3; /* Colore di sfondo */
    max-width: 70%; /* Larghezza massima */
    padding: 10px; /* Padding */
    border-radius: 10px; /* Bordi arrotondati */
    display: inline-block; /* Display inline-block */
    font-size: 16px; /* Dimensione del testo */
    text-align: left; /* Allineamento a sinistra */
    color: #302e2b; /* Colore del testo */
    overflow-wrap: break-word; /* Gestione del testo a capo */
}

/* Stile per l'input della chat */
.stChatInput {
    background-color: #ffffff; /* Colore di sfondo */
}

/* Stile per gli avatar */
.avatar {
    width: 30px; /* Larghezza */
    height: 30px; /* Altezza */
    border-radius: 50%; /* Bordi arrotondati */
    margin: 0 10px; /* Margini */
    align-self: flex-end; /* Allineamento a fine flessibile */
}
```

custom_theme.css

```
@import
url('https://fonts.googleapis.com/css2?family=Roboto:wght@300;400;700&display=swap'
);

/* Stile base per l'intera applicazione */
.stApp {
    background-color: #ffffff; /* Colore di sfondo della pagina */
    color: #000000; /* Colore del testo */
}

/* Stile per i pulsanti */
.stButton>button {
    background-color: #335C67; /* Colore di sfondo del pulsante */
    color: white; /* Colore del testo del pulsante */
    position: absolute; /* Posizionamento assoluto del pulsante */
    right: 50px; /* Distanza dal lato destro */
    top: -40px; /* Distanza dal lato superiore */
}

/* Stile per il titolo della pagina */
.stTitle {
    font-family: 'Roboto', sans-serif; /* Famiglia del font */
    font-size: 2.5em; /* Dimensione del font */
    color: #5e3a05; /* Colore del font */
    text-align: center; /* Allineamento del testo al centro */
    margin-bottom: 0; /* Nessun margine inferiore */
}

/* Stile per il box di risposta */
.st-ae {
    background-color: #f4f4f4; /* Colore di sfondo del box di risposta */
    color: #000000; /* Colore del testo */
    border-color: #202020; /* Colore del bordo */
}

/* Stile per il box di risposta quando attivo */
.st-ae:active {
```

```
    border-color: rgb(27, 66, 173); /* Colore del bordo quando selezionato */
}

/* Stile per l'intestazione */
.st-emotion-cache-12fmjuu {
    background-color: #ffffff; /* Colore di sfondo dell'intestazione */
}

/* Stile per il piè di pagina */
.st-emotion-cache-uhkwx6 {
    background-color: #ffffff; /* Colore di sfondo del piè di pagina */
}

/* Stile per i pulsanti personalizzati */
.customBtn {
    background-color: #f4f4f4; /* Colore di sfondo del pulsante */
    border-radius: 30%; /* Angoli arrotondati */
}

/* Stile per il testo dell'intestazione */
.stHeader {
    background-color: #335C67; /* Colore di sfondo */
    font-family: 'Roboto', sans-serif; /* Famiglia del font */
    font-size: 1.5em; /* Dimensione del font */
    color: #085457; /* Colore del testo */
}

/* Stile per l'espansore */
.stExpander {
    font-family: 'Roboto', sans-serif; /* Famiglia del font */
    font-size: 1em; /* Dimensione del font */
    color: #5e3a05; /* Colore del testo */
}

/* Stile per la barra laterale */
.stSideBar {
    font-family: 'Roboto', sans-serif; /* Famiglia del font */
```

```
    font-size: 1em; /* Dimensione del font */
    color: #5e3a05; /* Colore del testo */
    background-color: #f4f4f4; /* Colore di sfondo */
}

/* Stile per il testo Markdown */
.stMarkdown {
    font-family: 'Roboto', sans-serif; /* Famiglia del font */
    color: #335C67; /* Colore del testo */
}

/* Stile per oggetti chiave-valore */
.object-key-val {
    background-color: #f4f4f4; /* Colore di sfondo */
}

/* Stile personalizzato per l'intestazione */
.customHdr {
    color: #5e3a05; /* Colore del testo */
}

/* Regolazione della posizione per alcuni elementi */
.st-emotion-cache-7ym5gk {
    bottom: 20px; /* Posizione dal fondo */
}

/* Stile per lo sfondo della barra laterale */
.st-emotion-cache-6qob1r {
    background-color: #f4f4f4; /* Colore di sfondo */
}

/* Stile per i messaggi dell'utente */
.user-message {
    display: flex; /* Flexbox per l'allineamento */
    justify-content: flex-end; /* Allinea gli elementi a destra */
    align-items: center; /* Allinea gli elementi al centro */
    margin-bottom: 10px; /* Margine sotto il messaggio */
```

```css
}

/* Stile per i messaggi dell'assistente */
.assistant-message {
    display: flex; /* Flexbox per l'allineamento */
    justify-content: flex-start; /* Allinea gli elementi a sinistra */
    align-items: center; /* Allinea gli elementi al centro */
    margin-bottom: 10px; /* Margine sotto il messaggio */
    padding: 10px; /* Padding intorno al messaggio */
}

/* Stile generico per le bolle dei messaggi */
.stChatMessage {
    max-width: 70%; /* Larghezza massima della bolla */
    padding: 10px; /* Padding all'interno della bolla */
    border-radius: 10px; /* Angoli arrotondati */
    display: inline-block; /* Display inline-block */
    font-size: 16px; /* Dimensione del font */
}

/* Posizionamento della freccia di invio */
.eyeqlp51 {
    position: absolute; /* Posizionamento assoluto */
    top: 10px; /* Posizione dal lato superiore */
    left: -12px; /* Posizione dal lato sinistro */
}

/* Effetto hover per la freccia di invio */
.st-emotion-cache-1hkc054:hover {
    background-color: transparent; /* Sfondo trasparente */
    color: rgb(255,255,255); /* Colore bianco */
}

/* Stile per le bolle dei messaggi dell'utente */
.user-bubble {
    background-color: #335C67; /* Colore di sfondo */
    padding: 8px; /* Padding all'interno della bolla */
```

```
    border-radius: 10px 10px 0 10px; /* Angoli arrotondati con raggio specifico */
    text-align: right; /* Allinea il testo a destra */
    color: #ffffff; /* Colore del testo */
    max-width: 70%; /* Larghezza massima */
    overflow-wrap: break-word; /* Gestisce l'andata a capo del testo */
}

/* Stile per le bolle dei messaggi dell'assistente */
.assistant-bubble {
    background-color: #f4f4f4; /* Colore di sfondo */
    max-width: 70%; /* Larghezza massima */
    padding: 10px; /* Padding all'interno della bolla */
    border-radius: 10px 10px 10px 0; /* Angoli arrotondati con raggio specifico */
    text-align: left; /* Allinea il testo a sinistra */
    color: #302e2b; /* Colore del testo */
    overflow-wrap: break-word; /* Gestisce l'andata a capo del testo */
}

/* Stile per l'area di input della chat */
.stChatInput {
    background-color: #ffffff; /* Colore di sfondo */
}

/* Stile per il logo avatar */
.avatar {
    width: 30px; /* Larghezza dell'avatar */
    height: 30px; /* Altezza dell'avatar */
    border-radius: 50%; /* Forma circolare */
    margin: 0 10px; /* Margine intorno all'avatar */
    align-self: flex-end; /* Allinea in basso */
}
```

3.2 Alternativa open source

L’alternativa open source citata precedentemente che si affida alle librerie HuggingFace e Ollama in particolare si utilizza come modello di sentence-transformer "all-MiniLM-L6-v2" messo a disposizione dall'HuggingFace Hub e il modello “llama3” di Ollama. Per questo abbiamo creato un file HuggingFace_utility.py di cui mostriamo il codice commentato :

```
import streamlit as st
import torch
import gc
from langchain_community.embeddings import HuggingFaceEmbeddings
from langchain_community.vectorstores import FAISS
from langchain_community.chat_models import ChatOllama
from langchain.chains import ConversationalRetrievalChain
from langchain.memory import ConversationBufferWindowMemory
from googletrans import Translator
from functools import lru_cache


# Funzione per unire più file PDF in un unico file
def merge_pdfs(folder_path, output_path):
    merger = PdfMerger()  # Crea un oggetto PdfMerger per unire PDF

    for root, dirs, files in os.walk(folder_path):  # Itera attraverso tutti i file
nella cartella
        for file in files:
            if file.lower().endswith('.pdf'):  # Controlla se il file è un PDF
                file_path = os.path.join(root, file)
                merger.append(file_path)  # Aggiunge il file PDF al merger

    merger.write(output_path)  # Scrive il PDF unito nell'output specificato
    merger.close()  # Chiude il PdfMerger

# Funzione per leggere il contenuto testuale di un PDF
def read_pdf(file):
    document = ""  # Stringa vuota per memorizzare il testo estratto

    reader = PdfReader(file)  # Crea un oggetto PdfReader per leggere il PDF
    for page in reader.pages:
        document += page.extract_text()  # Estrae il testo da ogni pagina e lo
aggiunge al documento
        document = document.replace("\n-", "\n#")  # Sostituisce i trattini con i
cancelletto per coerenza

    return document  # Ritorna il testo estratto
```

```
# Funzione per suddividere un documento in segmenti più piccoli
def split_doc(document, chunk_size, chunk_overlap):
    splitter = RecursiveCharacterTextSplitter(
        separators="#",  # Usa il cancelletto come separatore
        chunk_size=chunk_size,  # Dimensione di ciascun segmento
        chunk_overlap=chunk_overlap  # Sovrapposizione tra segmenti adiacenti
    )
    split = splitter.split_text(document)  # Divide il documento in segmenti di
testo
    split = splitter.create_documents(split)  # Crea oggetti Document dai segmenti
di testo
    return split  # Ritorna i segmenti di testo come oggetti Document

# Funzione per creare embedding dei segmenti di testo e salvarli in un vector store
def embedding_storing(split, create_new_vs, existing_vector_store, new_vs_name):
    # Inizializza un modello di embedding utilizzando un modello pre-addestrato da
Hugging Face
    instructor_embeddings = HuggingFaceEmbeddings(model_name="sentence-
transformers/all-MiniLM-L6-v2")

    # Crea un vector store FAISS a partire dai segmenti di testo e dagli embedding
    db = FAISS.from_documents(split, instructor_embeddings)

    if create_new_vs:
        # Salva il vector store localmente con il nome specificato
        db.save_local("vector store/" + new_vs_name)
    else:
        # Carica un vector store esistente e lo unisce con il nuovo vector store
        load_db = FAISS.load_local("vector store/" + existing_vector_store,
instructor_embeddings, allow_dangerous_deserialization=True)
        load_db.merge_from(db)
        load_db.save_local("vector store/" + new_vs_name)

# Funzione per preparare il modello RAG conversazionale con una data configurazione
def prepare_rag_llm(vector_store, temperature, max_length):
    # Inizializza un modello di embedding da Hugging Face, specificando di usare la
CPU
    instructor_embeddings = HuggingFaceEmbeddings(
        model_name="sentence-transformers/all-MiniLM-L6-v2",
        model_kwargs={'device': 'cpu'}
    )

    # Carica il vector store FAISS localmente
    loaded_db = FAISS.load_local(f"vector store/{vector_store}",
instructor_embeddings, allow_dangerous_deserialization=True)
```

```
    # Configura il modello di linguaggio, specificando la temperatura e il numero
massimo di token previsti
    llm = ChatOllama(
        model="llama3",
        temperature=temperature,
        num_predict=max_length
    )

    # Configura un buffer di memoria per conservare le ultime 2 interazioni della
conversazione
    memory = ConversationBufferWindowMemory(
        k=2,
        memory_key="chat_history",
        output_key="answer",
        return_messages=True
    )

    # Configura una catena di elaborazione che combina un modello di linguaggio con
il modulo di recupero dei documenti
    qa_conversation = ConversationalRetrievalChain.from_llm(
        llm=llm,
        chain_type="stuff",  # Utilizza i primi 3 documenti recuperati per generare
la risposta
        retriever=loaded_db.as_retriever(search_kwargs={"k": 3}),
        return_source_documents=True,  # Restituisce anche i documenti sorgente
usati per generare la risposta
        memory=memory  # Utilizza il buffer di memoria configurato
    )

    return qa_conversation  # Ritorna la catena di elaborazione configurata

# Funzione per generare la risposta
#La funzione generate_answer è progettata per generare una risposta
#a una domanda utilizzando un modello di conversazione configurato in
st.session_state.conversation.
#La funzione verifica la presenza di un token, genera una risposta tramite il
modello,
#traduce la risposta dall'inglese all'italiano e restituisce sia la risposta
tradotta che le fonti dei documenti utilizzati per generarla.
#La funzione utilizza una cache LRU (Least Recently Used) per memorizzare fino a
128 risposte generate per migliorare le prestazioni.
@lru_cache(maxsize=128)
def generate_answer(question):
    # Inizializza la risposta predefinita e la fonte del documento
    answer = "Si è verificato un errore"
    doc_source = ["nessuna fonte"]
```

```
    # Genera la risposta usando il modello di conversazione
    response = st.session_state.conversation({"question": question})

    # Estrae la risposta dal response
    answer = response.get("answer").split("Risposta utile:")[-1].strip()

    # Estrae le fonti dei documenti dal response
    explanation = response.get("source_documents", [])
    doc_source = [d.page_content for d in explanation]

    # Traduttore per tradurre la risposta in italiano
    translator = Translator()
    translated_answer = translator.translate(answer, src='en', dest='it').text

    # Restituisce la risposta tradotta e le fonti dei documenti
    return translated_answer, doc_source
```

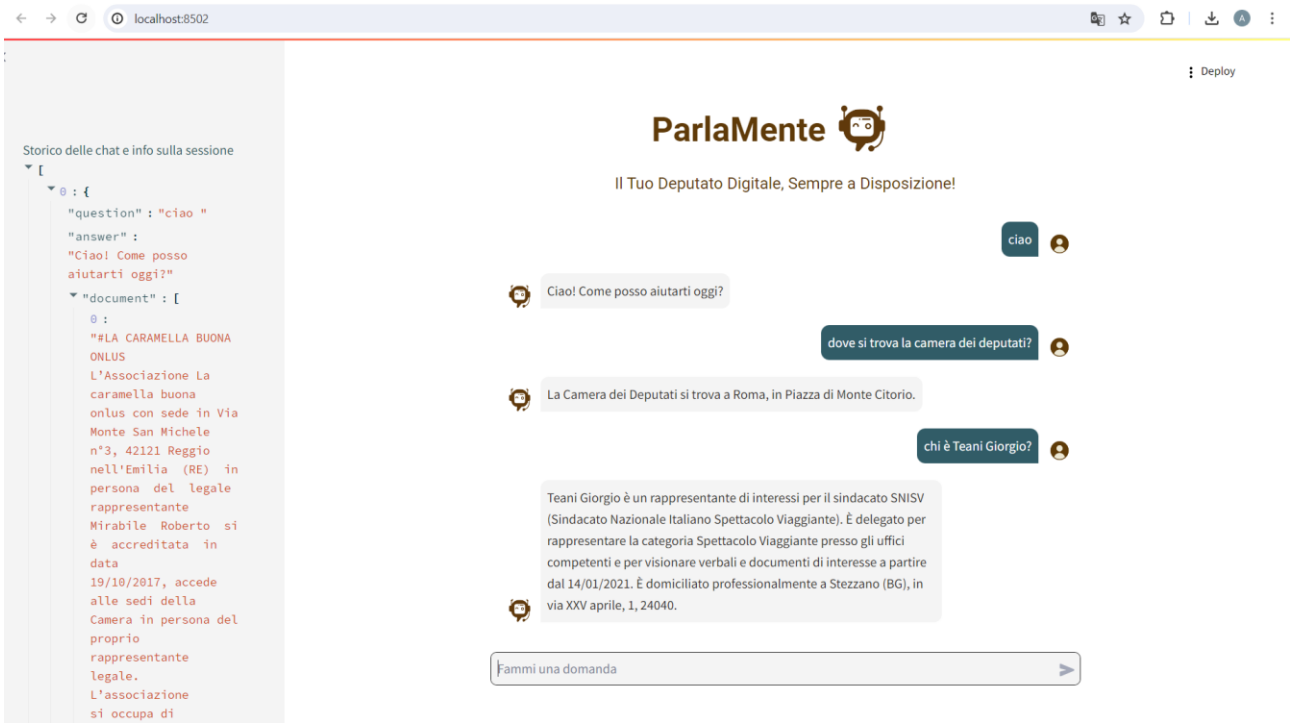
3.3 Parlamente : guida all'installazione ed esecuzione con esempi

Per installare il chatbot localmente sono stati effettuati questi passaggi:

1. Installazione di Anaconda: per creare l'ambiente di sviluppo;
2. Creazione di un environment: dopo l'installazione è stato creato un environment specifico usando Anaconda Navigator "textmining1" che va attivato sempre con il comando `conda activate textmining1` da terminale;
3. Installazione delle dipendenze: per installare tutte le dipendenze necessarie per il progetto descritte sopra sono stati lanciati i pip install indicati precedentemente ;
4. Configurazione di Visual Studio Code: dopo aver aperto la cartella contenente il codice del chatbot con un IDE come Visual Studio Code imposta l'interprete di Anaconda ;
5. Esecuzione del codice: se è la prima volta che esegui il chatbot, esegui lo script `merger.py`. Successivamente, per eseguire effettivamente il chatbot, esegui il seguente comando da terminale: streamlit run chatbot_parlamente.py . Dopo aver eseguito quest'ultimo comando, il chatbot verrà avviato e il browser si aprirà con l'indirizzo localhost.

```
(base) C:\Users\alema> conda activate textmining1

(textmining1) C:\Users\alema> cd C:\Users\alema\Desktop\chatbotParlamente\chatbotParlamente

(textmining1) C:\Users\alema\Desktop\chatbotParlamente\chatbotParlamente> streamlit run chatbot_parlamente.py

  You can now view your Streamlit app in your browser.

  Local URL: http://localhost:8502
  Network URL: http://192.168.1.5:8502
```

Seguono gli esempi di esecuzione con entrambi i temi :



Cliccare l’icona delle info per consultare la documentazione.