Antonio Di Geronimo
Riccardo Converso
Giuseppe Balzano

Progetto NLP 2024

di un sistema di Question Answering (QA) basato sul framework LangChain

Indice

Capitolo 1

LangChain

1.1 Cos'è LangChain

LangChain è una libreria open-source progettata per facilitare la costruzione di applicazioni basate su modelli di linguaggio di grandi dimensioni (LLM). Con l'espansione dell'uso dei LLM in vari settori, LangChain si propone come una soluzione versatile e modulare che consente agli sviluppatori di sfruttare al meglio le capacità di questi modelli. La libreria offre strumenti per la gestione e l'orchestrazione di diversi componenti di un'applicazione basata su LLM, includendo funzionalità per l'integrazione di dati, l'ottimizzazione delle prestazioni e la gestione dei flussi di lavoro.

LangChain si distingue per la sua flessibilità e per l'ampia gamma di funzionalità che supporta. Può essere utilizzato per una varietà di scopi, tra cui l'estrazione di informazioni, la generazione di testo, la traduzione automatica, e molto altro. La modularità della libreria consente agli sviluppatori di personalizzare e combinare i diversi componenti in base alle specifiche esigenze del progetto.

Per la realizzazione del progetto è stato importante identificare, analizzare e studiare gli strumenti forniti da LangChain in modo da poter implementare i punti richiesti:

- Automatic Speech Recognition (ASR)
- Natural Language Understanding (NLU)
- Knowledge Representation (KR)
- Decision Making (DM)
- Natural Language Generation (NLG)
- Text To Speech (TTS)

1.2 Automatic Speech Recognition (ASR)

LangChain consente la trascrizione audio grazie all'utilizzo di Google Speech-to-Text. Per far in modo di utilizzare i servizi Google, è stato necessario creare un progetto all'interno di Google Cloud e abilitarne l'API. Attraverso l'utilizzo della classe **SpeechToTextLoader**, è stato possibile trascrivere file audio con la **Google Cloud Speech-to-Text API** e caricare il testo trascritto nella cartella del progetto.

E' risultato necessario andare a modificare la lingua in modo da consentire al servizio di riconoscere le parole e andare a ricostruire le frasi in italiano:

```
language_codes=["it-IT"]
```

1.3 Natural Language Understanding (NLU)

I modelli di chat che LangChain mette a disposizione sono progettati per comprendere e generare risposte in linguaggio naturale, facilitando interazioni naturali e conversazioni intelligenti con gli utenti.

LangChain può integrare diversi modelli di linguaggio di grandi dimensioni da vari fornitori, come OpenAI (GPT-3, GPT-4), Google (BERT, T5), e altri modelli open-source e commerciali. Questa flessibilità consente agli sviluppatori di scegliere il modello più adatto alle loro esigenze specifiche.

Il modello da noi utilizzato è , definito nel seguente modo:

```
my_model = GoogleGenerativeAI(model="gemini-pro",
                              google_api_key=api_key)
```

Andando a definire un ulteriore parametro (**graph**) è possibile, attraverso il modello, generare query chyper in modo da interrogare il database a grafo utilizzato.

1.4 Knowledge Representation (KR)

Nel contesto dell'archiviazione e dell'analisi dei dati, la Knowledge Representation (KR) rappresenta un aspetto cruciale per organizzare e comprendere le informazioni in modo significativo. Neo4j, un avanzato database a grafo, offre una soluzione innovativa per la rappresentazione della conoscenza, grazie alla sua capacità di modellare dati complessi attraverso grafi.

Neo4j è progettato per gestire e interrogare dati interconnessi mediante una struttura di nodi, relazioni e proprietà. Questo modello di dati grafico si distingue per la sua capacità di rappresentare in modo naturale e intuitivo le relazioni tra entità, rendendo l'analisi delle connessioni e dei pattern molto più efficiente rispetto ai tradizionali modelli relazionali.

Nel nostro caso, abbiamo utilizzato un database denominato "Recommendations", fornito direttamente da Neo4j. Questo database include dettagli su film, generi cinematografici, attori coinvolti in ciascun film e i registi che li hanno diretti.

1.5 Decision Making (DM)

Questa sezione riguarda più la logica e le scelte implementative; non è quindi legata strettamente a LangChain. Affronteremo questo discorso più avanti.

1.6 Natural Language Generation (NLG)

L'NLG si occupa di prendere dati grezzi e trasformarli in testo naturale che possa essere facilmente compreso da un pubblico umano. Questa capacità di automatizzare la produzione di contenuti testuali non solo migliora l'efficienza nella creazione di documenti e report, ma consente anche una comunicazione più efficace e accessibile delle informazioni.

Nel nostro caso è stato definito un modello che, prese le risposte di query chyper fatte da un altro modello, genera una risposta in linguaggio naturale.

1.7 Text To Speech (TTS)

Il componente di Text To Speech (TTS) in LangChain rappresenta un elemento fondamentale per la trasformazione del testo scritto in output vocale. Questo processo consente di convertire testi digitali in audio parlato, facilitando una modalità di interazione più naturale e accessibile con i contenuti. E' risultato necessario, anche in questo caso, abilitare il servizio su Google Cloud in modo da utilizzare questa funzionalità. La classe utilizzata all'interno del codice è **TextToSpeechTool**. Per l'utilizzo della lingua italiana, è stato modificato il campo language_code e per utilizzare la voce maschile, è stato assegnato il valore "MALE" al campo ssml_gender:

```
language_code: str = "it-IT",
ssml_gender: Optional[texttospeech.SsmlVoiceGender] = 'MALE'
```

Capitolo 2

Realizzazione Progetto

2.1 Traccia

Si realizzi un sistema di Question Answering (QA) basato sul framework LangChain che includa i seguenti moduli:

- Automatic Speech Recognition (ASR)
- Natural Language Understanding (NLU)
- Knowledge Representation (KR)
- Decision Making (DM)
- Natural Language Generation (NLG)
- Text To Speech (TTS)

Come dominio si utilizzi il database d'esempio dei film fornito da Neo4j integrato con i testi descrittivi delle trame. Si utilizzi Neo4j per gli aspetti paradigmatici.

2.2 Automatic Speech Recognition (ASR)

Come primo passo è stato necessario far in modo di registrare, attraverso il microfono del computer, la voce della persona che interagisce con il chatbot. Questa operazione è stato possibile attraverso l'utilizzo della libreria **sounddevice**:

```
1    import sounddevice as sd
2    ...
3    stream = sd.InputStream(samplerate=sample_rate, channels=channels,
                dtype='int16', callback=callback)
```

Dopo aver creato l'oggetto stream, sono stati utilizzati i metodi **.start()** e **.stop()** per eseguire rispettivamente l'avvio della registrazione e l'interruzione della stessa.

Dopo esserci assicurati di aver salvato in locale la richiesta vocale dell'utente, si è passati alla conversione del vocale in testo.

```
    def process_audio(frame_param,widget_param):

        audio_data_np = np.concatenate(audio_data, axis=0)
        file_name = "registrazione.wav"
        write(file_name, sample_rate, audio_data_np)
        loader = SpeechToTextLoader(project_id=project_id, file_path =
                 file_name)
        docs = loader.load()
        question = docs[0].page_content
        msg_frame_bot.destroy()
        send(question,frame_param,widget_param)
```

Come mostrato nel codice soprastante, vengono inizialmente concatenati i campioni di audio presenti in **audio_data** attraverso il metodo **.concatenate()**. Il file audio viene convertito in testo utilizzando il **loader** di tipo **SpeechToTextLoader**, che crea un oggetto **docs** contenente la trascrizione. Una volta ottenuta la richiesta dell'utente, è possibile inoltrarla al modello per elaborarla attraverso il metodo **send()**.

2.3 Natural Language Understanding (NLU)

Per la realizzazione del chatbot abbiamo ritenuto opportuno utilizzare due differenti modelli in grado di interpretare la richiesta dell'utente. Il primo ha come obiettivo quello di interpretare la domanda dell'utente identificando le entità coinvolte e le relazioni tra di esse. Il secondo invece è responsabile della trasformazione dei dati strutturati (risposte ottenute da Neo4j) in testo comprensibile per l'utente.

Un primo modello è definito nel seguente modo:

```
    cypher_llm = GoogleGenerativeAI(model="gemini-pro", google_api_key=
                    api_key), graph=graph, verbose=True,
    cypher_llm_kwargs={"prompt": CYPHER_GENERATION_PROMPT, "memory":
                    readonlymemory, "verbose": False},
```

Tale modello prende la richiesta dell'utente e la trasforma in query cypher. L'obiettivo è quello di interrogare la banca dati e recuperare le informazioni storicizzate in essa. Per aiutare il modello a generare query che rispecchiassero le nostre esigenze, sono stati definiti dei prompt template nel seguente modo:

```
    CYPHER_GENERATION_PROMPT = PromptTemplate(
            input_variables = ["schema", "question", "chat_history"],
```

```
            template = CYPHER_GENERATION_TEMPLATE)
```

I prompt (**CYPHER_GENERATION_PROMPT**) sono realizzati a partire dai template (**CYPHER_GENERATION_TEMPLATE**). Vediamo un paio di esempi di template:

```
    # Quanti attori hanno recitato in Top Gun?
    MATCH (m:Movie {{title:"Top Gun"}})-[:ACTED_IN]-(person:Person)
    RETURN count(person) AS QUANTI_ATTORI_HANNO_RECITATO_IN_TOP_GUN

    # Chi ha recitato in Top Gun?
    MATCH (p:Person)-[:ACTED_IN]-(m:Movie)
    WHERE m.title = "Top Gun"
    RETURN p.name AS RECITATO_IN_TOP_GUN
```

Il secondo modello è stato utilizzato per aiutarci a tenere traccia delle preferenze dell'utente riguardo agli attori e ai generi di film. È stato definito nello stesso modo del modello precedente, con l'unica differenza nei template utilizzati. Vediamo ora un esempio di un template che storicizza le informazioni sulle preferenze:

```
    #Consigliami un film di Tom Hanks
    MATCH (u:User)
    WHERE u.name = "User"
    MATCH (p:Person)
    WHERE p.name = "Tom Hanks"
    MERGE (u)-[r:LIKES]->(p)
    ON CREATE SET r.count = 1
    ON MATCH SET r.count = r.count + 1
```

Quando viene richiesto un consiglio su un film con un determinato attore, se non esiste già, viene creato un collegamento tra l'utente e quell'attore. A ogni successiva richiesta di consiglio sullo stesso attore, viene aggiornato tale collegamento aumentando un contatore di uno. Tale contatore è un attributo che appartiene alla relazione [:LIKES] che collega lo User con un particolare film o attore.

2.3.1 Definizione delle "chain"

Le catene si riferiscono a sequenze di chiamate, che siano a un LLM, a uno strumento o a una fase di pre-elaborazione dei dati. Il modo principale supportato per farlo è con LCEL.

LangChain Expression Language, o LCEL, è un modo dichiarativo per comporre facilmente catene insieme. LCEL è stato progettato sin dal primo giorno per supportare la messa in produzione di prototipi, senza modifiche al codice , dalla più semplice catena "prompt + LLM" alle catene più complesse (abbiamo visto persone eseguire con successo catene LCEL con centinaia di passaggi in produzione).

Per l'implementazione del chatbot, abbiamo avuto l'esigenza di implementare due chain differenti:

- Una prima catena composta sia dal modello che prende la frase in linguaggio naturale e genera ed esegue una query cypher ed un secondo modello che prende il risultato della query e lo esprime in linguaggio naturale.

```
1    chain = GraphCypherQAChain.from_llm(
2        cypher_llm=GoogleGenerativeAI(model="gemini-pro",
             google_api_key=api_key), graph=graph, verbose=True,
3        cypher_llm_kwargs={"prompt": CYPHER_GENERATION_PROMPT,
              "memory": readonlymemory, "verbose": False},
4        qa_llm=GoogleGenerativeAI(model="gemini-pro",
                google_api_key=api_key),
5        qa_llm_kwargs={"prompt": QA_GENERATION_PROMPT, "memory":
                 readonlymemory}, memory=memory)
6
```

- la seconda, invece, ha il compito di effettuare query cypher nel caso in cui l'utente esprima preferenze su un determinato attore o un genere in particolare.

```
1    chain_user = GraphCypherQAChain.from_llm(
2                 GoogleGenerativeAI(model="gemini-pro",
3                 google_api_key=api_key), graph=graph,
4                 verbose=True,
5       cypher_llm_kwargs={"prompt": CYPHER_USER_GENERATION_PROMPT,
6                 "verbose": False})
7
```

2.3.1.1 Utilizzo del metodo invoke()

Il metodo invoke è tipicamente utilizzato per eseguire l'intera catena su un input dato. Nel nostro caso viene passato in input la richiesta dell'utente in modo da essere elaborata.

```
def invoke_model(...):
    result = {'result': "Nessuna risposta disponibile"}
    try:
        result = chain.invoke({"query": question})
        print(f"Final answer: {result['result']}")
    except Exception as e:
        result['result'] = "Scusa penso di non aver capito, puoi
                ripetere?"
        print(f"Final answer: {result['result']}")
    add_message(frame, result['result'], "bot", message_frame_bot)
    try:
        question = "Domanda: " + question
        chain_user.invoke({"query": question})
    except Exception as e:
        print("")
```

2.3.2 Gestione della memoria della conversazione

Una parte molto importante e necessaria per lo sviluppo della conversazione è il mantenimento in memoria di tutte le interazioni avvenute fino a quel momento. Per realizzare ciò è stata implementata ed utilizzata una vera e propria memoria che ricordasse sia le richieste fatte dall'utente che le risposte del chatbot.

Con lo scopo di dare indicazioni al modello su come utilizzare tale storia della conversazione, è stato importante inserire all'interno del template indicazioni su come comportarsi. Tali indicazioni sono le seguenti:

```
CYPHER_GENERATION_TEMPLATE = """Task:Continua la seguente conversazione
    dove Human rappresentano le mie domande e AI rappresentano le tue
   risposte: {chat_history}
    Human: {question}
    I messaggi più in basso sono i più recenti, quindi i più importanti
   .
    Devi rispondere generando una dichiarazione Cypher per interrogare
   un database grafico.

    ...
"""
```

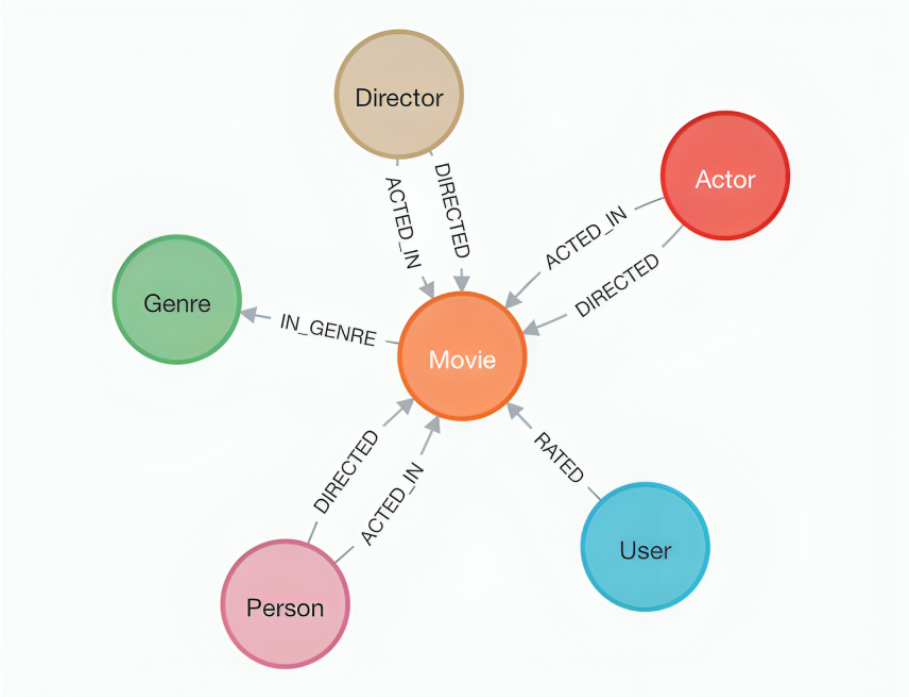
Sono state definite anche due tipi di memoria: una **ConversationBufferMemory** e l'altra **ReadOnlySharedMemory**.
La prima è un tipo di memoria dinamica che memorizza e gestisce le conversazioni in corso. È progettato per essere aggiornato continuamente con nuovi messaggi e per mantenere un contesto temporale delle interazioni recenti.
La seconda, invece, è un tipo di memoria immutabile, destinata alla condivisione di informazioni fisse tra componenti diversi. Una volta popolata, non può essere modificata.

2.4 Knowledge Representation (KR)

La Knowledge Representation (KR) è il processo di strutturazione e organizzazione delle informazioni in un formato che consenta una comprensione e un'interazione efficaci da parte di sistemi informatici. Neo4j, un database a grafi, è particolarmente adatto per la KR grazie alla sua capacità di rappresentare e gestire relazioni complesse tra entità in modo naturale e intuitivo.

Per i nostri scopi, è stato utilizzato un database fornito da Neo4j chiamato "**Recommendations**". Questo database include dettagli su film, generi cinematografici, attori coinvolti in ciascun film e i registi che li hanno diretti.
Sono presenti sei differenti entità: Actor, Director, Genre, Movie, Person, Movie, User. Sono presenti anche differenti tipi di relazioni: ACTED_IN, DIRECTED, IN_GENRE, LIKES, RATED.

Vediamo di seguito riportata la struttura del database:



2.5 Decision Making (DM)

L'obiettivo di questa sezione è stato quello di consentire al chatbot di fornire suggerimenti basati sia sulle nostre preferenze sia su richieste specifiche. In questa fase, è stato fondamentale definire alcune scelte progettuali per indirizzare le decisioni del chatbot in modo mirato.

Una delle scelte riguarda il suggerimento di film specificando un attore. Quando l'utente fa questa richiesta, il chatbot deve prima recuperare su Neo4j il genere preferito dall'utente e successivamente trovare i film di quell'attore appartenenti a quel genere.

La stessa cosa è stata fatta quando viene richiesto un film specificando il genere. Viene recuperato l'attore preferito dall'utente e successivamente vengono consigliati i film in base all'attore e al genere.

Di seguito è riportato un esempio della prima scelta menzionata:

```
Se viene richiesto un consiglio o un suggerimento su un film PARTENDO
    DA UN ATTORE, per esempio, CONSIGLIAMI UN FILM CON TOM HANKS,
    esegui questa query:


    MATCH (u:User)
    WHERE u.name = "User"
    OPTIONAL MATCH (u)-[r:LIKES]->(g1:Genre)
    WITH u, g1, COALESCE(SUM(r.count), 0) AS direct_count
    OPTIONAL MATCH (u)-[r:LIKES]->(p:Person)-[:ACTED_IN]->(m:Movie)-[:
    IN_GENRE]->(g2:Genre)
    WITH u, g1, direct_count, g2, COALESCE(SUM(r.count), 1) AS
    indirect_count
    WITH g1 AS genre, direct_count, SUM(indirect_count) AS
    total_indirect_count
    WITH genre.name AS genre_name, direct_count + total_indirect_count
    AS total_likes
    ORDER BY total_likes DESC
    LIMIT 1
    MATCH (m:Movie)-[:IN_GENRE]->(genre)
    WHERE genre.name = genre_name
    MATCH (p:Person)-[:ACTED_IN]->(m)
    WHERE p.name = "Tom Hanks"
    RETURN m.title AS TITOLO_FILM
    ORDER BY RAND()
    LIMIT 1
```

2.6 Natural Language Generation (NLG)

Per la generazione di testo in linguaggio naturale abbiamo previsto che il chatbot sia in grado di rispondere all'utente riformulando il risultato della query al database. Il codice corrispondente è il seguente:

```
    QA_GENERATION_TEMPLATE = """Task: Genera una risposta completa per
    rispondere a {question} usando TUTTI i risultati contenuti in {
    context}, senza escluderne nessuno
    """

```

```
    ...

    QA_GENERATION_PROMPT = PromptTemplate(
                    input_variables=["question", "context",
                    "chat_history"],template=QA_GENERATION_TEMPLATE)

    ...

    qa_llm=GoogleGenerativeAI(model="gemini-pro", google_api_key=
            api_key),
    qa_llm_kwargs={"prompt": QA_GENERATION_PROMPT, "memory":
            readonlymemory}, memory=memory
```

Com'è possibile notare da codice, è importante utilizzare la variabile **context** come paramtero di **QA_GENERATION_PROMPT** per far in modo di prendere in considerazione il risultato della query ed elaborarlo.

Una strategia che abbiamo ritenuto opportuno utilizzare è stata quella di assegnare un alias alla risposta della query cypher in modo da conservare informazioni sulla risposta stessa e quindi, di conseguenza, anche sulla richiesta fatta. Se ad esempio viene chiesto "Chi ha recitato in Top Gun?", l'alias associato al risultato sarà "RECITATO_IN_TOP_GUN".

```
# Chi ha recitato in Top Gun?
    MATCH (p:Person)-[:ACTED_IN]-(m:Movie)
    WHERE m.title = "Top Gun"
    RETURN p.name AS RECITATO_IN_TOP_GUN
```

2.7 Text To Speech (TTS)

Il Text To Speech è quella funzionalità che ci consente di utilizzare una voce digitale per comunicare con l'utente. Una volta che il primo modello recupera le informazioni dal database e le trasforma in linguaggio naturale, occorre che una voce digitale "legga" questa risposta. Ciò favorisce l'interazione tra chatbot e utente.

```
def play_response(response):
    speech_file = tts.run(response)
    pygame.mixer.init()
    pygame.mixer.music.load(speech_file)
    pygame.mixer.music.play()
    while pygame.mixer.music.get_busy():
        continue
    pygame.mixer.music.stop()
    pygame.mixer.quit()
    os.remove(speech_file)
```

La funzione **play_response** utilizza la sintesi vocale e la libreria **pygame** per generare e riprodurre una risposta audio. La funzione accetta un testo come input, lo converte in un file audio tramite un modulo Text-to-Speech (**tts.run(response)**), e utilizza pygame per la riproduzione del file audio. Il modulo mixer di pygame viene inizializzato, il file audio viene caricato e riprodotto. Durante la riproduzione, un ciclo while mantiene il programma in attesa fino a che l'audio non termina. Una volta finita la riproduzione, le risorse del mixer vengono liberate e il file audio temporaneo viene rimosso dal sistema.

2.8 Modo alternativo di interagire con il chatbot

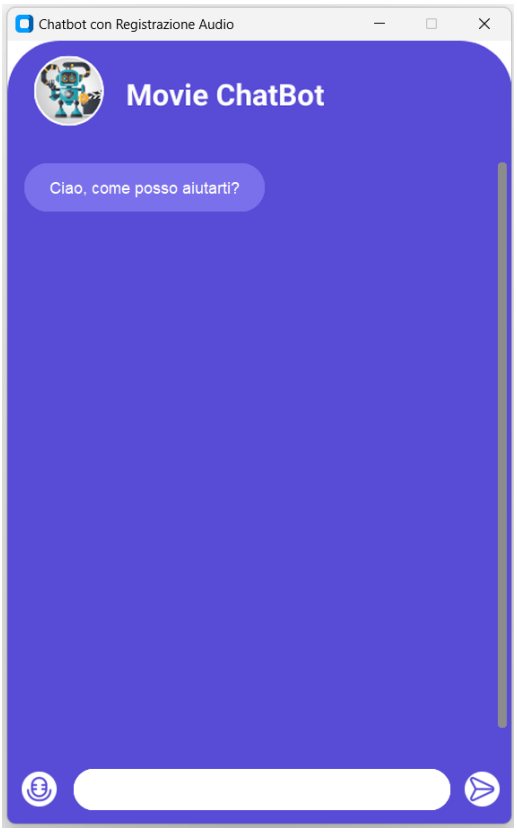
Per inviare richieste al chatbot, oltre all'uso del linguaggio parlato, è possibile inserire testo tramite tastiera. Il processo di generazione della risposta del chatbot segue le stesse modalità descritte nelle sezioni precedenti, con l'eccezione della fase di trasformazione della voce in testo.

Capitolo 3

Interfaccia Grafica

3.1 Interazione con il ChatBot

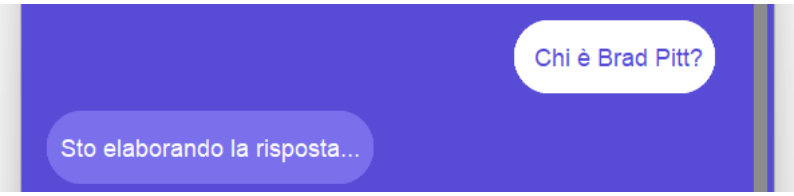
Per poter interagire con il chatbot è stata realizzata un'interfaccia grafica con l'utilizzo della libreria **CustomTkinter**. Questa libreria ci ha concesso di realizzare in modo semplice e veloce l'interfaccia grafica, mantenendo il template iniziale che avevamo stabilito. L'interfaccia si presenta come segue:



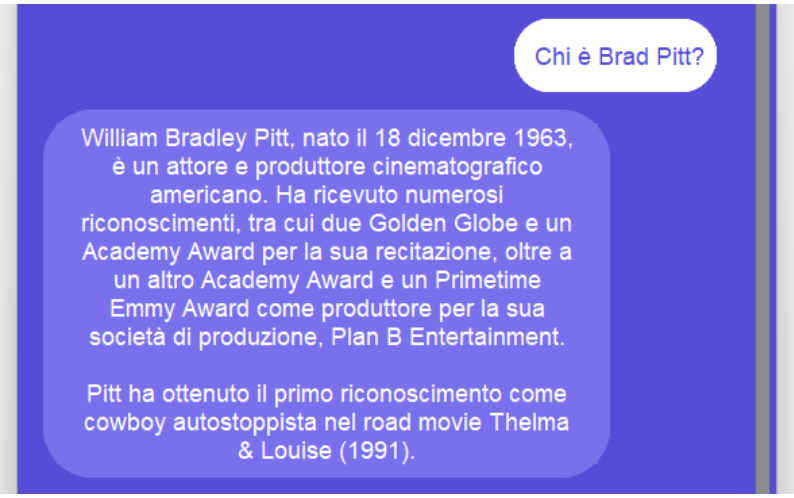
Nella parte inferiore sono presenti due pulsanti: uno con l'icona del microfono, che permette di interagire vocalmente con il chatbot, e uno con l'icona di un aeroplanino di carta, che consente di inviare il messaggio scritto nella sezione a destra del pulsante stesso.

Quando si sottomette una richiesta vocale (tenendo premuta l'icona del microfono), comparirà a schermo un messaggio da parte del chatbot che dice "Sto ascoltando...".

Una volta terminato il messaggio vocale, comparirà tra i messaggi dell'utente la richiesta (es. *"Chi è Brad Pitt?"*) e tra i messaggi del bot, prima della risposta alla richiesta, un messaggio contenente la scritta "Sto elaborando la risposta..."



Una volta elaborata la richiesta, la risposta comparirà tra i messaggi del bot.



Una funzionalità implementata per migliorare l'interazione con il bot consiste nella possibilità di ascoltare le risposte alle nostre richieste tramite una voce artificiale, generata convertendo il testo scritto in audio. Occorrerà quindi cliccare sul messaggio generato per far partire l'audio della risposta.