



Titolo abbreviato come intestazione



UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

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

Progetto per il corso di Text Mining

Realizzazione di un ChatBot con tecnologia RAG

Anno Accademico 2023/2024

Candidati:

Nike Di Giacomo **matr.** M63001641

Matteo Della Ventura **matr.** -

Andrea Di Fusco **matr.** -

Gennaro Napolano **matr.** M63001659

Abstract

Ormai i chatbot hanno acquistato rilevanza grazie al sempre più popolare ChatGPT. Questi però esistevano già in precedenza, anche se le loro risposte erano molto limitate sia a livello di fluidità del testo restituito che alla loro limitata conoscenza. Ad oggi ormai tutti abbiamo usato ChatGPT per qualsiasi domanda, anche la più sciocca. Ma se volessimo avere un nostro chatbot che risponde ai nostri dati? Qui il discorso si fa più interessante. Infatti, anche se ChatGPT ha una conoscenza veramente vasta grazie ai dati su cui è stato addestrato (praticamente tutto il web) ci potrebbero essere alcuni argomenti che non conosce nel dettaglio. Ad esempio, se fossimo in un'azienda e avessimo dei documenti privati o dei regolamenti interni che non sono mai stati divulgati sul web nel caso in cui volessimo estrarre alcune informazioni da queste risorse come potremmo fare? In questo caso dovremo realizzare un chatbot che usa tali documenti come risorsa di conoscenza, ma che si basa su modelli di generazione del linguaggio naturale al fine di fornire risposte sensate e fluide.

Indice

Abstract	i
1 Introduzione	1
1.1 Tecnologie Utilizzate	2
1.1.1 Retrieval-Augmented Generation	3
1.1.2 API di OpenAI	3
1.1.3 Anaconda	3
1.1.4 Visual Studio Code	4
1.1.5 Streamlit	4
1.1.6 Ollama	5
1.2 Stato iniziale del progetto	6
1.2.1 bot.py	7
1.2.2 chatbot_streamlit_combined.py	9
1.2.3 falcon.py	12
1.2.4 main_bot.py	14
2 Modifiche al Progetto - <i>Parlibot</i>	17
2.1 Preparazione.py	18
2.2 parlibot.py	24
2.3 Versione con API di OpenAI	34

<i>INDICE</i>	iii
3 Guida all'installazione ed esecuzione	36
3.1 Installazione ed esecuzione in locale	36
3.1.1 Guida all'installazione con OpenAI	40
3.2 Esempio di utilizzo	40
4 Conclusioni	42

Chapter 1

Introduzione

I chatbot sono diventati fondamentali in vari settori, tra cui l'assistenza clienti, il marketing e i servizi informativi, grazie alla loro capacità di interagire con gli utenti in modo efficace e personalizzato. Ma cosa sono realmente?

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 è la loro abilità di interazione linguistica. Sono in grado di comprendere e rispondere al linguaggio umano, permettendo così di risolvere domande, fornire informazioni e assistere in compiti specifici. I chatbot più avanzati utilizzano l'intelligenza artificiale (IA) e tecnologie di machine learning per affinare la comprensione del contesto e delle intenzioni dell'utente. È facile, allora, intuire che creare un chatbot da zero è un compito piuttosto difficile. 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, e creare un'interfaccia utente facile da usare. Inoltre, è necessario migliorare continuamente il sistema per garantire risposte precise e pertinenti, il che richiede tempo e risorse considerevoli. Si procederà quindi con cautela, iniziando a presentare gli strumenti e le tecnologie utilizzati per la realizzazione del chatbot ParliBot, per poi illustrare lo stato iniziale del progetto.

1.1 Tecnologie Utilizzate

Nella creazione del progetto, sono stati impiegati una serie di strumenti e tecnologie fondamentali, ciascuno svolgendo un ruolo cruciale nel processo di sviluppo del chatbot. Grazie a questi strumenti, è stato possibile accelerare il processo di sviluppo e implementare facilmente funzionalità avanzate nel chatbot, garantendo un'esperienza utente di alta qualità e un'interazione fluida e naturale con l'utente finale. Di seguito si riportano i dettagli.

Durante tutta la lavorazione del progetto, per la modellazione e l'analisi del linguaggio naturale, si è fatto largo uso di tecnologie avanzate come il modello di linguaggio a rappresentazione generativa (**RAG**) e i modelli di linguaggio a lungo termine (**LLM**).

1.1.1 Retrieval-Augmented Generation

Il modello **RAG**, si è dimostrato particolarmente utile nella gestione delle conversazioni generative, consentendo al chatbot di comprendere il contesto della conversazione e generare risposte coerenti e contestualmente rilevanti. Allo stesso modo, i modelli di linguaggio a lungo termine (LLM) hanno contribuito a migliorare la capacità del chatbot di comprendere e generare testi coerenti e informativi, fornendo una base solida per la creazione di conversazioni fluide e naturali.

1.1.2 API di OpenAI

Sono state sfruttate le **API di OpenAI**, che offrono una vasta gamma di servizi e funzionalità per lo sviluppo di chatbot avanzati. Queste API forniscono accesso a potenti modelli di linguaggio, come GPT (Generative Pre-trained Transformer), che costituiscono il nucleo del chatbot *ParliBot*. Grazie alle API di OpenAI, è stato possibile integrare facilmente i modelli di linguaggio all'interno del chatbot, consentendo di sfruttarne la potenza e l'efficacia senza dover implementare complessi algoritmi di machine learning da zero.

1.1.3 Anaconda

Anaconda si è rivelato uno strumento indispensabile per la gestione dell'ambiente di sviluppo e la gestione 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. Si riporta nel *capitolo 3* una dettagliata guida all'installazione delle librerie necessarie.

1.1.4 Visual Studio Code

Inoltre, **Visual Studio Code** è stato utilizzato per la scrittura del codice e la gestione del progetto, offrendo un ambiente di sviluppo intuitivo e ricco di funzionalità per la programmazione in Python.

1.1.5 Streamlit

Streamlit, è una libreria open-source che offre un'interfaccia semplice e intuitiva per la creazione di applicazioni web interattive in Python.

Streamlit si è rivelato uno strumento prezioso per la creazione dell'interfaccia utente del chatbot, consentendo di visualizzare in modo chiaro e intuitivo le conversazioni e le risposte generate dal sistema.

Attraverso Streamlit, è stato possibile creare un'interfaccia web interattiva in poche righe di codice, consentendo agli utenti di interagire con il chatbot direttamente dal proprio browser senza la necessità di installare alcun software aggiuntivo. Questo ha reso il chatbot facilmente accessibile e utilizzabile da qualsiasi dispositivo connesso a Internet, ampliando così la sua portata e la sua utilità. Inoltre, Streamlit offre una

serie di funzionalità avanzate per la personalizzazione dell'interfaccia utente, consentendo di integrare facilmente elementi grafici, tabelle e widget interattivi per migliorare l'esperienza dell'utente e rendere la navigazione più intuitiva e coinvolgente.

1.1.6 Ollama

Infine è stato utilizzato **Ollama**. Essa è una piattaforma progettata per facilitare l'uso e l'implementazione di modelli di intelligenza artificiale, in particolare quelli di linguaggio come GPT. Fornisce strumenti per sviluppare, addestrare e implementare modelli AI, gestendo l'intero ciclo di vita del modello, dall'addestramento alla distribuzione. Inoltre, permette l'accesso e la gestione dei dati necessari per l'addestramento, offrendo anche un'interfaccia utente intuitiva per interagire con i modelli. Ollama 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.

1.2 Stato iniziale del progetto

Al di là delle tecnologie utilizzate, è ora possibile avviare il processo di creazione del chatbot, muovendo i primi passi nell'implementazione del progetto. Durante la fase di progettazione e di realizzazione di ParliBot, lo stato iniziale è costituito da un programma python articolato in quattro moduli principali che collaborano per creare una piattaforma di chatbot basata su documenti personalizzati.

I quattro moduli del programma sono: **bot.py**, **falcon.py**, **main_bot.py** e **chatbot_streamlit_combined.py**.

In particolare, il programma offre una piattaforma di chatbot che utilizza documenti caricati dagli utenti come base di conoscenza. Gli utenti caricano documenti tramite la interfaccia di bot.py, che vengono processati per generare embeddings. Questi embeddings vengono salvati e utilizzati per addestrare un chatbot configurato tramite main_bot.py. Il modulo chatbot_streamlit_combined.py gestisce la interfaccia utente complessiva e le diverse pagine dell'applicazione, mentre falcon.py fornisce le funzioni di backend per la gestione dei documenti e del modello di linguaggio. Di seguito verranno illustrate le funzionalità di ciascun modulo, riportando esclusivamente estratti di codice essenziali alla creazione del progetto finale, integrando aspetti teorici e considerazioni sull'utilizzo attuale delle tecnologie coinvolte.

1.2.1 bot.py

Questo modulo 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 possono specificare il modello di embeddings, la dimensione dei chunk e l'overlap tra essi, e scegliere se creare un nuovo archivio di vettori o aggiornare uno esistente. Una volta caricati, i documenti vengono processati, suddivisi in chunk, e gli embeddings vengono generati e salvati utilizzando il modulo `falcon.py`. 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 apprendimento profondo per generare queste rappresentazioni, che sono cruciali per il funzionamento del chatbot.

```
1 with st.form("document_input"):  
2     document = st.file_uploader(  
3         "Knowledge Base Documents", type=['pdf', 'txt'],  
4         help=".pdf or .txt file",  
5         accept_multiple_files=True  
6     )  
7     row_1 = st.columns([2, 1, 1])
```

```
6     with row_1[0]:
7         instruct_embeddings = st.text_input(
8             "Model Name of the Instruct Embeddings",
9             value="hkunlp/instructor-xl"
10        )
11    with row_1[1]:
12        chunk_size = st.number_input(
13            "Chunk Size", value=200, min_value=0, step=1,
14        )
15    with row_1[2]:
16        chunk_overlap = st.number_input(
17            "Chunk Overlap", value=10, min_value=0,
18            step=1,
19            help="higher than chunk size"
20        )
21
22    if save_button:
23        # Read the uploaded file
24        if document.name[-4:] == ".pdf":
25            document = falcon.read_pdf(document)
26        elif document.name[-4:] == ".txt":
27            document = falcon.read_txt(document)
28        else:
29            st.error("Check if the uploaded file is .pdf or
30                    .txt")
31
32    # Split document
```

```

30     split = falcon.split_doc(document, chunk_size,
                               chunk_overlap)
31
32     # Check whether to create new vector store
33     create_new_vs = None
34     if existing_vector_store == "<New>" and new_vs_name
        != "":
35         create_new_vs = True
36     elif existing_vector_store != "<New>" and
        new_vs_name != "":
37         create_new_vs = False
38     else:
39         st.error(
40             """Check the 'Vector Store to Merge the
               Knowledge'
41             and 'New Vector Store Name'"""
42         )
43
44     # Embeddings and storing
45     falcon.embedding_storing(
46         split, create_new_vs, existing_vector_store,
        new_vs_name
47     )

```

1.2.2 chatbot_streamlit_combined.py

Questo modulo gestisce la selezione e l'inizializzazione delle diverse pagine dell'applicazione Streamlit, incluse quelle dedicate all'embedding

dei documenti e al chatbot RAG (Retrieval-Augmented Generation). RAG combina il recupero di informazioni da una base di conoscenza con la generazione di risposte pertinenti. Questo approccio migliora significativamente la pertinenza e l'accuratezza delle risposte, rendendo i chatbot basati su RAG particolarmente efficaci per applicazioni che richiedono una comprensione approfondita e contestuale dei contenuti.

Il modulo include anche funzioni per la gestione della memoria GPU, assicurando la disponibilità di risorse sufficienti per l'esecuzione del modello di linguaggio su GPU. Inoltre, offre un'interfaccia utente per configurare il modello di LLM (Large Language Model) e le impostazioni del chatbot, gestendo la sessione di conversazione e visualizzando la cronologia delle chat e le fonti dei documenti. LLM, come GPT-3 e GPT-4, sono modelli di intelligenza artificiale che utilizzano reti neurali profonde per comprendere e generare linguaggio naturale. Questi modelli sono addestrati su vasti corpus di testo, consentendo loro di apprendere le strutture grammaticali, i significati delle parole e il contesto in modo dettagliato.

```
1 # Prepare the LLM model
2 if "conversation" not in st.session_state:
3     st.session_state.conversation = None
4 if token:
5     st.session_state.conversation =
        falcon.prepare_rag_llm(
```

```
6         token, existing_vector_store, temperature,
           max_length
7     )
8
9     # Chat history
10    if "history" not in st.session_state:
11        st.session_state.history = []
12
13    # Source documents
14    if "source" not in st.session_state:
15        st.session_state.source = []
16
17    # Display chats
18    for message in st.session_state.history:
19        with st.chat_message(message["role"]):
20            st.markdown(message["content"])
21
22    # Ask a question
23    if question := st.chat_input("Ask a question"):
24        # Append user question to history
25        st.session_state.history.append({"role": "user",
26                                         "content": question})
27        # Add user question
28        with st.chat_message("user"):
29            st.markdown(question)
30
31    # Answer the question
```



```
31     answer, doc_source =  
        falcon.generate_answer(question, token)  
32     with st.chat_message("assistant"):  
33         st.write(answer)  
34     # Append assistant answer to history  
35     st.session_state.history.append({"role":  
        "assistant", "content": answer})  
36  
37     # Append the document sources  
38     st.session_state.source.append({"question":  
        question, "answer": answer, "document":  
        doc_source})
```

1.2.3 falcon.py

Questo modulo contiene le principali funzioni per la gestione dei documenti e del modello di linguaggio. Inoltre, gestisce la generazione delle risposte alle domande degli utenti mediante il modello di linguaggio e gli embeddings dei documenti caricati. La tecnologia RAG (Retrieval-Augmented Generation) combinata con LLM offre un potente strumento per migliorare l'interazione tra esseri umani e macchine. I chatbot basati su queste tecnologie possono fornire risposte più accurate e contestualizzate, migliorando l'esperienza utente in molteplici settori, tra cui l'assistenza clienti, l'educazione, la sanità e il commercio elettronico. La possibilità di personalizzare questi modelli con documenti specifici li rende strumenti potenti per applicazioni aziendali, dove la

pertinenza delle risposte basate su dati aziendali specifici è fondamentale.

```
1 # Create the chatbot
2 qa_conversation = ConversationalRetrievalChain.from_llm(
3     llm=llm,
4     chain_type="stuff",
5     retriever=loaded_db.as_retriever(search_kwargs={"k":
6         3}),
7     return_source_documents=True,
8     memory=memory,
9 )
10
11 def generate_answer(question, token):
12     answer = "An error has occurred"
13     if token == "":
14         answer = "Insert the Hugging Face token"
15         doc_source = ["no source"]
16     else:
17         response =
18             st.session_state.conversation({"question":
19                 question})
20         answer = response.get("answer").split("Helpful
21             Answer:")[-1].strip()
22         explanation = response.get("source_documents",
23             [])
24         doc_source = [d.page_content for d in
25             explanation]
```

```
21     return answer, doc_source
```

1.2.4 main_bot.py

Il modulo consente agli utenti di configurare il modello di LLM, selezionare l'archivio di vettori e impostare parametri come la temperatura e la lunghezza massima delle risposte. Gestisce la sessione di conversazione e visualizza la cronologia delle chat e le fonti dei documenti utilizzati per rispondere alle domande.

```
1 # Setting the LLM
2 with st.expander("Setting the LLM"):
3     st.markdown("This page is used to have a chat with
4         the uploaded documents")
5     with st.form("setting"):
6         row_1 = st.columns(3)
7         with row_1[0]:
8             token = st.text_input("Hugging Face Token",
9                 type="password")
10
11         with row_1[1]:
12             llm_model = st.text_input("LLM model",
13                 value="tiiuae/falcon-7b-instruct")
14
15         with row_1[2]:
16             instruct_embeddings =
17                 st.text_input("Instruct Embeddings",
```

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

14
15     row_2 = st.columns(3)
16     with row_2[0]:
17         vector_store_list = os.listdir("vector
18             store/")
19         default_choice = (
20             vector_store_list.index('naruto_snake')
21             if 'naruto_snake' in vector_store_list
22             else 0
23         )
24         existing_vector_store = st.selectbox("Vector
25             Store", vector_store_list, default_choice)
26
27     with row_2[1]:
28         temperature = st.number_input("Temperature",
29             value=1.0, step=0.1)
30
31     with row_2[2]:
32         max_length = st.number_input("Maximum
33             character length", value=300, step=1)
34
35     create_chatbot = st.form_submit_button("Create
36         chatbot")
37
38     # Prepare the LLM model
39     if "conversation" not in st.session_state:
```

```
36     st.session_state.conversation = None
37
38     if token:
39         st.session_state.conversation =
            falcon.prepare_rag_llm(
40             token, existing_vector_store, temperature,
                max_length
41         )
```

Chapter 2

Modifiche al Progetto - *Parlibot*

ParliBot è un chatbot interattivo basato su un modello di Retrieval-Augmented Generation (RAG), progettato per rispondere a domande basate su documenti PDF uniti e indicizzati. Questo progetto utilizza tecnologie di NLP avanzate per fornire risposte accurate e contestuali utilizzando dati provenienti da documenti PDF caricati e processati. Per lo sviluppo del chatbot sono stati realizzati due script Python: "**preparazione.py**" e "**parlibot.py**". Il primo, è stato realizzato al fine di effettuare il merge dei PDF componenti il dataset assegnato e effettuare embedding creando il vector store utilizzando dei modelli di apprendimento automatico per estrarre e memorizzare rappresentazioni vettoriali del contenuto del PDF. Il secondo file, consiste invece nel chatbot vero e proprio, contiene quindi le chiamate alle API uti-

lizzate al fine di generare risposte alle domande degli utenti sulla base del documento PDF fornito.

2.1 Preparazione.py

Questo script è utile per chiunque debba gestire e analizzare grandi quantità di documenti PDF, specialmente in ambito di ricerca o documentazione. Permette di combinare, leggere, suddividere e analizzare documenti PDF in modo efficiente, utilizzando tecniche avanzate di elaborazione del linguaggio naturale.

Di seguito sono riportati e spiegati i dettagli del codice implementato.

Per quanto riguarda le librerie utilizzate sono tali per cui:

```
1 import os
2 import torch
3 import gc
4 from pypdf import PdfReader
5 from langchain.text_splitter import
    RecursiveCharacterTextSplitter
6 from langchain_community.embeddings import
    HuggingFaceEmbeddings
7 from langchain_community.vectorstores import FAISS
8 from PyPDF2 import PdfMerger
```

- **os**: Modulo per interagire con il sistema operativo.
- **torch**: Libreria utilizzata qui per gestire la memoria della GPU.

- **gc**: Libreria per la gestione della garbage collection in Python.
- **pypdf**: Libreria per la gestione dei file PDF.
- **langchain.text_splitter**: Utilizzato per suddividere il testo in blocchi (chunks).
- **langchain_community.embeddings**: Utilizzato per creare embedding con modelli pre-addestrati.
- **langchain_community.vectorstores**: Utilizzato per memorizzare e gestire gli embedding.
- **PyPDF2**: Utilizzato per gestire l'unione dei file PDF.

```

1 def merge_pdfs_from_folder(folder_path, output_path):
2     merger = PdfMerger()
3     for root, dirs, files in os.walk(folder_path):
4         for file in files:
5             if file.lower().endswith('.pdf'):
6                 file_path = os.path.join(root, file)
7                 print(f'Adding {file_path}')
8                 merger.append(file_path)
9     merger.write(output_path)
10    merger.close()
11    print(f'Merged PDF saved to {output_path}')

```

La funzione sovrastante, percorre ricorsivamente una cartella il cui path è specificato come parametro di input dello stesso metodo, aggiungendo tutti i file PDF trovati in un unico file PDF di output. La

funzione permette di navigare la cartella e tutte le sue sottocartelle.

```
1 def clear_memory():
2     torch.cuda.empty_cache()
3     gc.collect()
```

La funzione sovrastante libera la cache della GPU e forza la garbage collection per liberare la memoria.

```
1 def read_pdf(file):
2     document = ""
3     reader = PdfReader(file)
4     for page in reader.pages:
5         document += page.extract_text()
6     return document
```

La funzione sovrastante legge il contenuto di un file PDF, passatogli in input, pagina per pagina e restituisce il testo estratto come una stringa.

```
1 def split_doc(document, chunk_size, chunk_overlap):
2     splitter = RecursiveCharacterTextSplitter(
3         chunk_size=chunk_size,
4         chunk_overlap=chunk_overlap
5     )
6     split = splitter.split_text(document)
7     split = splitter.create_documents(split)
8     return split
```

La funzione sovrastante suddivide il documento in blocchi di testo utilizzando un algoritmo di suddivisione ricorsiva, tenendo conto delle dimensioni dei blocchi e della sovrapposizione tra di essi. Effettua quindi l'operazione di **split** del documento.

```

1 def embedding_storing(split, create_new_vs,
    existing_vector_store, new_vs_name):
2     instructor_embeddings =
        HuggingFaceEmbeddings(model_name="sentence-transformers/all-M
        model_kwargs={'device': 'cpu'}) # Usa la CPU
3
4     db = FAISS.from_documents(split,
        instructor_embeddings)
5
6     vector_store_path = "vector_store/" + new_vs_name
7     if create_new_vs:
8         if not os.path.exists(vector_store_path):
9             os.makedirs(vector_store_path)
10            db.save_local(vector_store_path)
11            print(f"Vector store saved to
                {vector_store_path}")
12    else:
13        load_db = FAISS.load_local(
14            vector_store_path,
15            instructor_embeddings,
16            allow_dangerous_deserialization=True
17        )

```

```

18         load_db.merge_from(db)
19         load_db.save_local(vector_store_path)
20         print(f"Vector store merged and saved to
                {vector_store_path}")
21
22     print("Il documento è stato salvato.")

```

La funzione crea embedding del testo suddiviso utilizzando un modello di embedding di **HuggingFace** e li memorizza in un archivio vettoriale **FAISS**. Se l'archivio vettoriale esiste già, lo aggiorna con i nuovi dati.

```

1 def main():
2     folder_path =
3         '/Users/nikedigiacomo/Desktop/Parlamento'
4     output_path =
5         '/Users/nikedigiacomo/Desktop/Prova/prova.pdf'
6     vector_store_name = "nome_vector_store"
7
8     merge_pdfs_from_folder(folder_path, output_path)
9
10    vector_store_path = f"vector_store/" +
11        vector_store_name
12    if os.path.exists(vector_store_path):
13        for file in os.listdir(vector_store_path):
14            file_path = os.path.join(vector_store_path,
15                                    file)
16            if os.path.isfile(file_path):

```

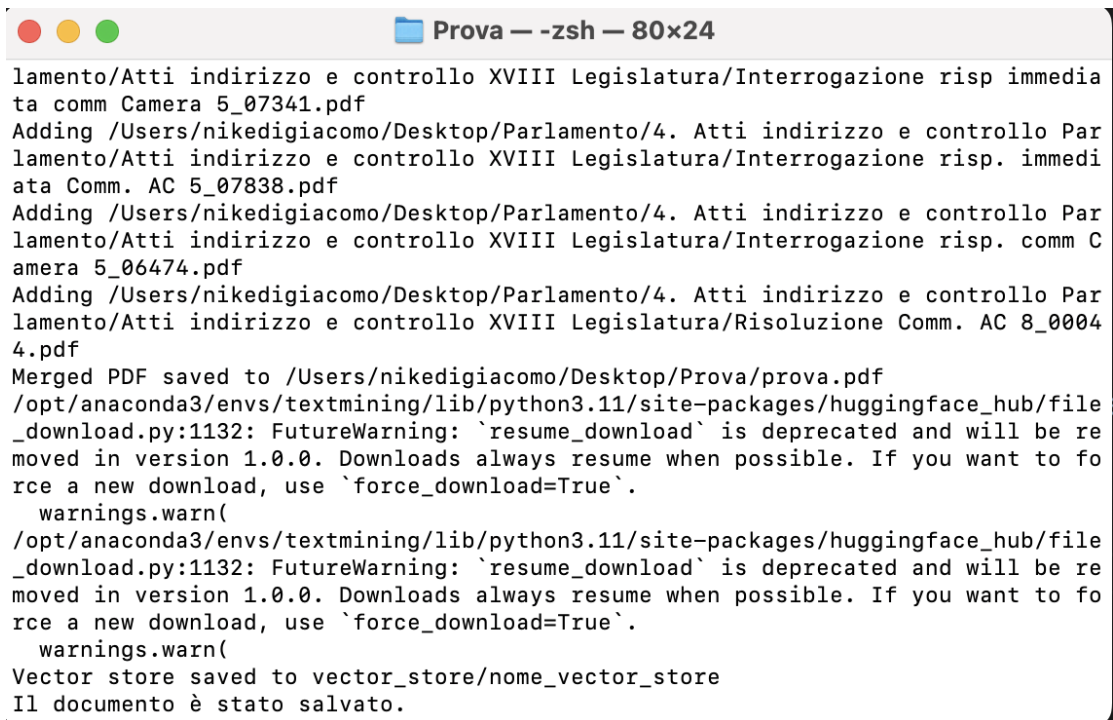
```

13         os.unlink(file_path)
14         os.rmdir(vector_store_path)
15
16     document = read_pdf(output_path)
17     split = split_doc(document, chunk_size=500,
18                        chunk_overlap=50)
19     embedding_storing(split, create_new_vs=True,
20                      existing_vector_store="",
21                      new_vs_name=vector_store_name)
22
23 if __name__ == "__main__":
24     main()

```

La funzione sovrastante rappresenta il main del programma implementato. Essa segue l'intero flusso di lavoro: unisce i PDF, cancella l'archivio vettoriale esistente, legge il PDF unito, lo suddivide in blocchi, crea e memorizza gli embedding in un nuovo archivio vettoriale. Come è possibile notare sono stati specificati dei **path assoluti** per la directory contenente i file PDF da unire e per specificare dove salvare il file PDF generato dalla funzione di **merge**.

Una volta eseguito lo script Python il risultato sarà la creazione del vector store e del file PDF, come riportato di seguito:



```

lamento/Atti indirizzo e controllo XVIII Legislatura/Interrogazione risp immedia
ta comm Camera 5_07341.pdf
Adding /Users/nikedigiaco/Desktop/Parlamento/4. Atti indirizzo e controllo Par
lamento/Atti indirizzo e controllo XVIII Legislatura/Interrogazione risp. immedi
ata Comm. AC 5_07838.pdf
Adding /Users/nikedigiaco/Desktop/Parlamento/4. Atti indirizzo e controllo Par
lamento/Atti indirizzo e controllo XVIII Legislatura/Interrogazione risp. comm C
amera 5_06474.pdf
Adding /Users/nikedigiaco/Desktop/Parlamento/4. Atti indirizzo e controllo Par
lamento/Atti indirizzo e controllo XVIII Legislatura/Risoluzione Comm. AC 8_0004
4.pdf
Merged PDF saved to /Users/nikedigiaco/Desktop/Prova/prova.pdf
/opt/anaconda3/envs/textmining/lib/python3.11/site-packages/huggingface_hub/file
_download.py:1132: FutureWarning: `resume_download` is deprecated and will be re
moved in version 1.0.0. Downloads always resume when possible. If you want to fo
rce a new download, use `force_download=True`.
  warnings.warn(
/opt/anaconda3/envs/textmining/lib/python3.11/site-packages/huggingface_hub/file
_download.py:1132: FutureWarning: `resume_download` is deprecated and will be re
moved in version 1.0.0. Downloads always resume when possible. If you want to fo
rce a new download, use `force_download=True`.
  warnings.warn(
Vector store saved to vector_store/nome_vector_store
Il documento è stato salvato.

```

Figure 2.1: Creazione PDF

2.2 parlibot.py

Il codice è uno script Python che utilizza Streamlit per creare un'interfaccia web per un chatbot basato su RAG. Questo script è utile per creare un chatbot avanzato che può rispondere a domande basate su **documenti pre-caricati**. Il chatbot permette agli utenti di porre domande e ottenere risposte, utilizzando modelli di embedding per il recupero di informazioni e un modello di linguaggio per generare risposte. Le librerie utilizzate sono le seguenti:

```
1 import streamlit as st
```

```
2 import torch
3 import gc
4 from langchain_community.embeddings import
    HuggingFaceEmbeddings
5 from langchain_community.vectorstores import FAISS
6 from langchain_community.chat_models import ChatOllama
7 from langchain.chains import ConversationalRetrievalChain
8 from langchain.memory import
    ConversationBufferWindowMemory
9 from googletrans import Translator
10 from functools import lru_cache
```

- **Streamlit (st):** Libreria per la creazione di applicazioni web interattive.
- **torch:** Libreria per il machine learning, utilizzata per gestire la memoria della GPU.
- **gc:** Libreria per la gestione della garbage collection in Python.
- **langchain_community.embeddings:** Modulo per creare embedding con modelli pre-addestrati.
- **langchain_community.vectorstores:** Modulo per memorizzare e gestire gli embedding.
- **langchain_community.chat_models:** Modulo per utilizzare modelli di linguaggio per la generazione di risposte.

- **langchain.chains:** Modulo per creare catene di elaborazione per il recupero di informazioni e la generazione di risposte.
- **langchain.memory:** Modulo per gestire la memoria della conversazione.
- **googletrans:** Libreria per la traduzione automatica.
- **functools.lru_cache:** Decoratore per la memorizzazione in cache dei risultati delle funzioni.

La funzione sottostante libera la cache della GPU e forza la garbage collection per liberare la memoria.

```

1 def clear_memory():
2     torch.cuda.empty_cache()
3     gc.collect()

```

La funzione sottostante prepara il modello RAG per il chatbot. Crea embedding utilizzando **HuggingFaceEmbeddings** e carica poi un archivio vettoriale FAISS contenente gli embedding. Successivamente inizializza un modello di linguaggio (si è scelto **ChatOllama** con il modello **llama3**), configura poi la memoria della conversazione per mantenere uno storico delle interazioni ed infine crea una catena di recupero e generazione, la **ConversationalRetrievalChain**, che utilizza il modello di linguaggio e l'archivio vettoriale per rispondere alle domande dell'utente.

```

1 def prepare_rag_llm(token, vector_store_list,
    temperature, max_length):
2     instructor_embeddings = HuggingFaceEmbeddings(
3         model_name="sentence-transformers/all-MiniLM-L6-v2",
4         model_kwargs={'device': 'cpu'}
5     )
6
7     loaded_db = FAISS.load_local(
8         f"vector_store/{vector_store_list}",
9         instructor_embeddings,
10        allow_dangerous_deserialization=True
11    )
12
13    llm = ChatOllama(model="llama3")
14
15    memory = ConversationBufferWindowMemory(
16        k=2,
17        memory_key="chat_history",
18        output_key="answer",
19        return_messages=True,
20    )
21
22    qa_conversation =
23        ConversationalRetrievalChain.from_llm(
24            llm=llm,
25            chain_type="stuff",
26            retriever=loaded_db.as_retriever(search_kwargs={"k":
27                3}),

```



```

26         return_source_documents=True,
27         memory=memory
28     )
29
30     return qa_conversation

```

La funzione **generate_answer** genera una risposta alla domanda dell'utente. Essa utilizza una cache per memorizzare le risposte delle domande frequenti. Se il token di Hugging Face non è fornito, restituisce un messaggio di errore. Il metodo utilizza la catena di conversazione per ottenere una risposta e le fonti dei documenti traduce infine la risposta in italiano utilizzando googletrans.

```

1 @lru_cache(maxsize=128)
2 def generate_answer(question, token):
3     answer = "$\u+FFFF\u2014rificato un errore"
4     doc_source = ["nessuna fonte"]
5
6     if token == "":
7         answer = "Inserisci il token Hugging Face"
8     else:
9         response =
10             st.session_state.conversation({"question":
11                 question})
12         answer = response.get("answer").split("Risposta
13             utile:")[-1].strip()
14         explanation = response.get("source_documents",

```

```

    [])
12     doc_source = [d.page_content for d in
                    explanation]
13
14     translator = Translator()
15     translated_answer = translator.translate(answer,
        src='en', dest='it').text
16
17     return translated_answer, doc_source

```

La funzione **main** che esegue lo script. In primo luogo libera la memoria chiamando `clear_memory()`. Definisce lo stile della pagina utilizzando **HTML** e **CSS**. Inizializza inoltre, lo stato della sessione per la conversazione, la cronologia e le fonti dei documenti. Prepara il modello RAG se il token di Hugging Face è fornito. Visualizza poi la cronologia della chat. Il main consente la gestione dell'input dell'utente per le domande e genera risposte infine permette la visualizzazione delle risposte e le fonti dei documenti.

```

1 def main():
2     clear_memory()
3
4     st.image("/Users/nikedigiacomo/Desktop/logo.png",
        width=200)
5
6     st.markdown("""
7     <style>

```

```
8      @import url('https://fonts.googleapis.com/css2?
9      family=Roboto:wght@300
10     ;400;700&display=swap');
11     .main {
12         background-color: #e8f5e9;
13         color: #2e7d32;
14     }
15     .stButton>button {
16         background-color: #66bb6a;
17         color: white;
18     }
19     .stTitle {
20         font-family: 'Roboto', sans-serif;
21         font-size: 2.5em;
22         color: #1b5e20;
23         text-align: center;
24         margin-bottom: 0;
25     }
26     .stHeader {
27         font-family: 'Roboto', sans-serif;
28         font-size: 1.5em;
29         color: #1b5e20;
30     }
31     .stExpander {
32         font-family: 'Roboto', sans-serif;
33         font-size: 1em;
34         color: #1b5e20;
35     }
```

```

36     .stMarkdown {
37         font-family: 'Roboto', sans-serif;
38         color: #2e7d32;
39     }
40     .stChatMessage {
41         background-color: #ffffff;
42         padding: 10px;
43         border-radius: 10px;
44         margin-bottom: 10px;
45         border: 1px solid #cccccc;
46     }
47     .stChatInput {
48         background-color: #ffffff;
49     }
50 </style>
51 """ , unsafe_allow_html=True)
52
53 st.markdown("<h1 class='stTitle'>ParliBot
[U+FFFD] [U+FFFD]\"", unsafe_allow_html=True)
54
55 st.markdown("""
56 <div style='text-align: center; margin-bottom:
30px;'>
57     <p style='font-family: "Roboto", sans-serif;
font-size: 1.2em; color: #1b5e20;'>
58         Benvenuto nel Chatbot RAG! Interagisci con
il Deputato Digitale!
59     </p>

```

```

60     </div>
61     """ , unsafe_allow_html=True)
62
63     if "conversation" not in st.session_state:
64         st.session_state.conversation = None
65     if "history" not in st.session_state:
66         st.session_state.history = []
67     if "source" not in st.session_state:
68         st.session_state.source = []
69
70     HUGGING_FACE_TOKEN = "nike"
71     VECTOR_STORE = "nome_vector_store"
72     TEMPERATURE = 1.0
73     MAX_LENGTH = 300
74
75     if HUGGING_FACE_TOKEN:
76         st.session_state.conversation = prepare_rag_llm(
77             HUGGING_FACE_TOKEN, VECTOR_STORE,
78             TEMPERATURE, MAX_LENGTH
79         )
80     st.subheader("Cronologia della Chat")
81     for message in st.session_state.history:
82         with st.chat_message(message["role"]):
83             st.markdown(f'<div
84
85                 class="stChatMessage">{message["content"]}</div>',
86                 unsafe_allow_html=True)

```

```

85     question = st.chat_input("Fai una domanda")
86     if question:
87         st.session_state.history.append({"role": "user",
88                                         "content": question})
89         with st.chat_message("user"):
90             st.markdown(f'<div
91                         class="stChatMessage">{question}</div>',
92                         unsafe_allow_html=True)
93
94         with st.spinner('Sto elaborando la tua
95                         risposta...'):
96             answer, doc_source =
97                 generate_answer(question,
98                                 HUGGING_FACE_TOKEN)
99
100        with st.chat_message("assistant"):
101            st.markdown(f'<div
102                        class="stChatMessage">{answer}</div>',
103                        unsafe_allow_html=True)
104
105        st.session_state.history.append({"role":
106                                        "assistant", "content": answer})
107        st.session_state.source.append({"question":
108                                        question, "answer": answer, "document":
109                                        doc_source})
110
111        with st.expander("Cronologia della chat e
112                        informazioni sui documenti sorgente"):

```

```
101         st.write(st.session_state.source)
102
103 if __name__ == "__main__":
104     main()
```

2.3 Versione con API di OpenAI

Come esposto precedentemente, il chatbot utilizza come modello di linguaggio "mistral" di "Ollama". Volendo invece adottare come modello di linguaggio "GPT" è necessario utilizzare le API di "OpenAI". Per prima cosa bisogna creare un file .env in cui inserire come variabile di ambiente la API key di OpenAI, fondamentale se si vogliono utilizzare le API di OpenAI.

Di seguito sono riportate, invece, le modifiche al codice del chatbot:

```
1     import openai
2     from dotenv import load_dotenv
3     import os
```

- **openai**: modulo fondamentale per utilizzare le API di OpenAI.
- **dotenv**: modulo contenente le funzioni per caricare le variabili di ambiente dal file .env.
- **os**: modulo contenente le funzioni per interagire con il sistema operativo.

```

1  # Carica le variabili d'ambiente dal file .env
2  load_dotenv()
3  openai_api_key = os.getenv('OPENAI_API_KEY')
4
5  # Imposta la chiave API di OpenAI
6  openai.api_key = openai_api_key

```

Le istruzioni sovrastanti servono per caricare la variabile d'ambiente dal file `.env`, memorizzare la OpenAI key in una variabile locale e infine impostarla.

Nel codice originale, la preparazione del modello LLM locale era necessaria perché stavamo utilizzando un modello ospitato localmente (HuggingFace) e un database di embedding (FAISS) per implementare un sistema RAG. Tuttavia, passando all'uso delle API di OpenAI, non è più necessario gestire un modello locale, poiché l'inferenza del modello è gestita interamente dai server di OpenAI. Questo semplifica notevolmente il codice e riduce i requisiti di risorse locali.

```

1  # Funzione per preparare il modello RAG (non necessaria
    con OpenAI)
2  def prepare_rag_llm():
3      return None # Non è necessario preparare un
        modello locale

```

Le modifiche successive al codice del chatbot riguardano semplicemente la rimozione delle parti in cui si chiedeva il token di "HuggingFace", non essendo più necessario in questo caso.

Chapter 3

Guida all'installazione ed esecuzione

3.1 Installazione ed esecuzione in locale

Per l'installazione del chatbot è possibile seguire i seguenti passi, che consentiranno di avere questa tecnologia localmente.

1. Per prima cosa è necessario installare Anaconda, al seguente link:
Installazione di Anaconda.
2. In secondo luogo, una volta terminata l'installazione, creeremo un **environment** specifico, per proseguire all'installazione dei pacchetti che saranno necessari ai fini del nostro progetto.

L'environment, può essere creato facilmente grazie alla funzione

Create di **Anaconda Navigator**, come riportato nell'immagine sottostante.

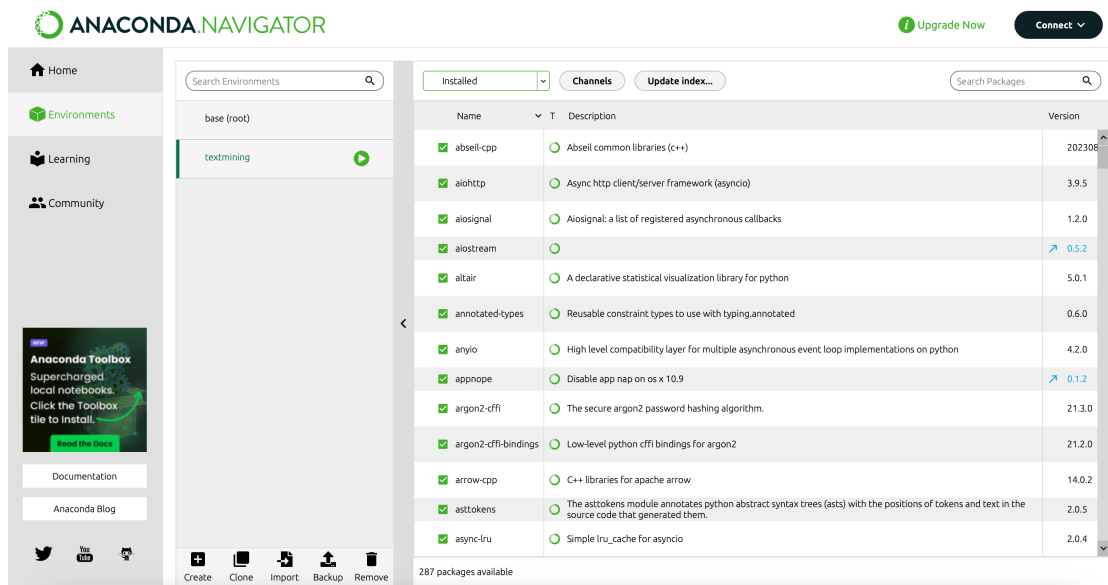


Figure 3.1: Environment dedicato

Si è scelto di creare un ambiente chiamato "textmining", quest'ultimo dovrà essere attivato mediante il comando "conda activate textmining" da terminale, affinché si passi dall'environment base a quello specificato.

- Il terzo passo consiste nell'installare tutte le dipendenze necessarie per l'esecuzione dello script, è possibile farlo nel seguente modo:

```
1 pip install lanchain_community
```

```

2     pip install streamlit
3     pip install PyPDF
4     pip install PyPDF2

```

4. Il quarto passo, sarà aprire mediante un IDE la cartella contenente il codice, si è scelto Visual Studio Code. Una volta aperta la cartella di nostro interesse, sarà necessario impostare manualmente l'interprete di Anaconda nel seguente modo:

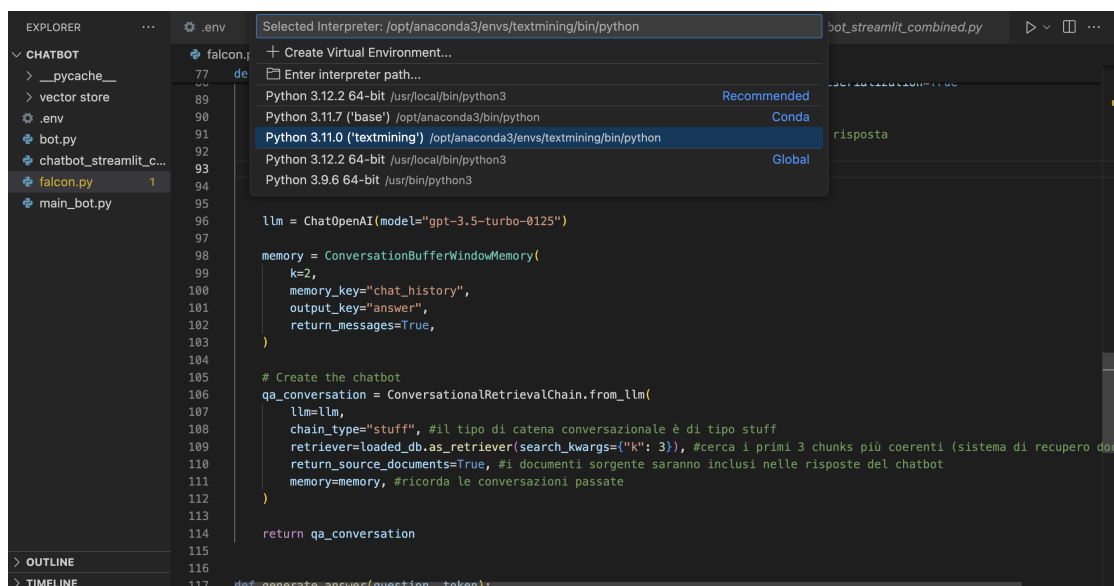


Figure 3.2: Apertura dell'interprete in VS Code

5. Per l'installazione di **Ollama** è possibile scaricare lo script d'installazione al seguente link:

Installazione di Ollama.

6. Per eseguire il codice del chatbot nel caso in cui fosse la prima esecuzione, si dovrà per prima cosa creare un **Nuovo Terminale** e digitare da tastiera il comando:

```
,
1 python3 run preparazione.py
```

Il comando sovrastante è necessario per l'esecuzione dello script Python che consente il merge dei PDF presenti nella cartella locale "Parlamento" e per la creazione dell'embedding e dunque del vector store.

Nota: questo passo è necessario soltanto se è la prima volta che si utilizza il chatbot, dato che l'ambiente è statico. Successivamente, si utilizzerà il seguente comando per il run dello script "parlibot.py" che consente l'esecuzione del del chatbot vero e proprio lanciando la "web-app".

```
,
1 Streamlit run parlibot.py
```

Eseguito quest'ultimo comando, da terminale si avrà il seguente risultato, e ci sarà poi l'apertura da browser dell'indirizzo local host.

```
(textmining) nikedigiacom@MacBook-Air-di-Nike Prova % Streamlit run chatbot.py ]
You can now view your Streamlit app in your browser.

Local URL: http://localhost:8501
Network URL: http://192.168.1.4:8501
```

Figure 3.3: Esecuzione in local host

3.1.1 Guida all'installazione con OpenAI

Quando detto sopra, resta valido nel caso in cui si usasse la versione del chatbot che utilizza le API di OpenAI. Le differenze sostanziali sono le seguenti:

1. In primo luogo sarà necessario, essendo esse API a pagamento caricare del credito sul proprio profilo di OpenAI per proseguire alla creazione di una **API key** e per permetterne l'utilizzo. Questa chiave dovrà essere salvata e non condivisa essendo essa segreta.
2. Sarà necessario installare, la libreria "openai" per permettere di utilizzare le API offerte da OpenAI, che possono essere usate grazie all'utilizzo di una API Key, che è stata cablata nel codice del chatbot. ,

```
1 pip install openai
```

Per quanto riguarda l'esecuzione, i passaggi sono **invariati**

3.2 Esempio di utilizzo

Di seguito si riporta un esempio di utilizzo del chatbot. Per l'esecuzione sono stati eseguiti tutti i passaggi della sezione Guida all'installazione ed esecuzione.

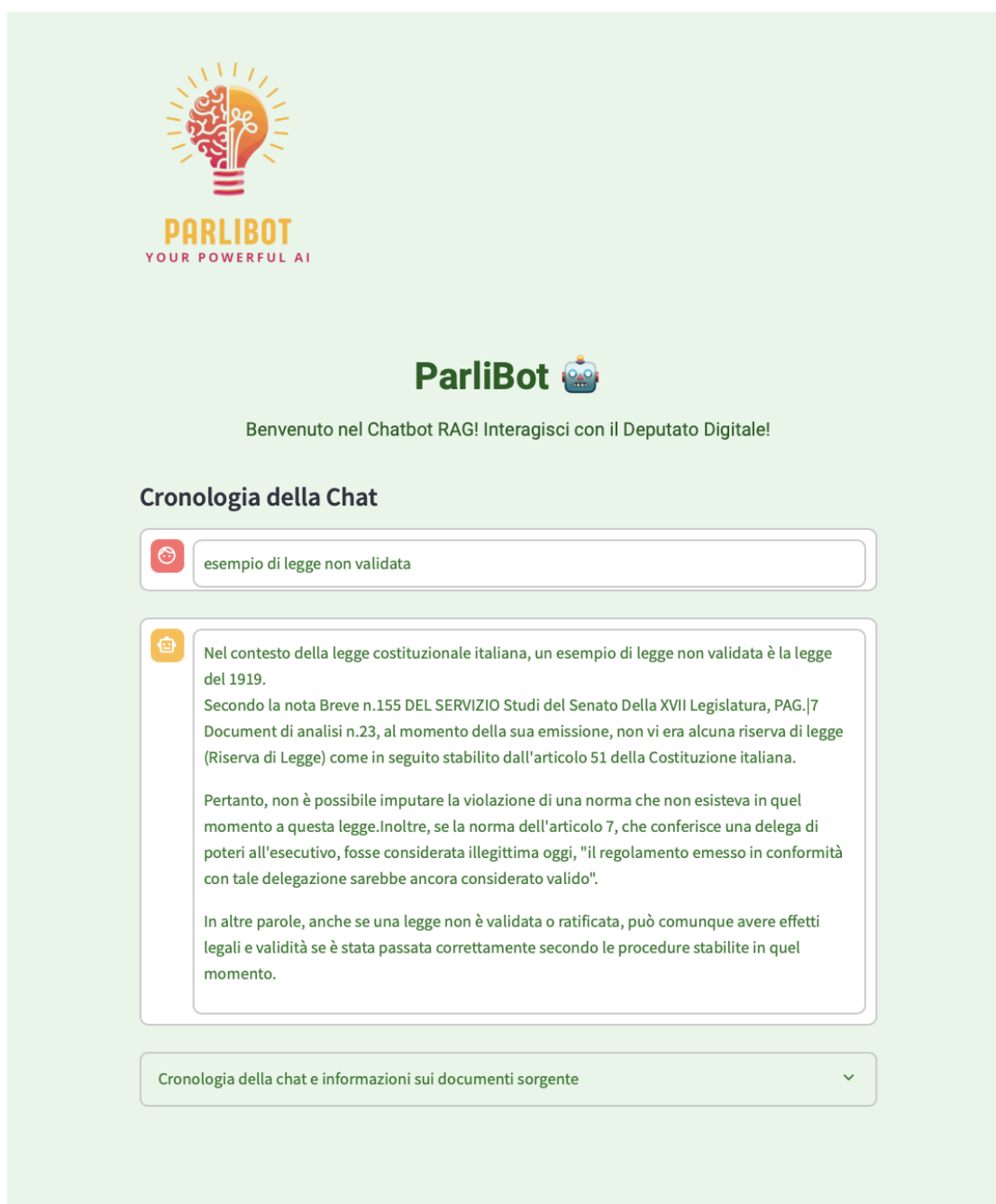


Figure 3.4: Environment dedicato

Chapter 4

Conclusioni

Donec et nisl id sapien blandit mattis. Aenean dictum odio sit amet risus. Morbi purus. Nulla a est sit amet purus venenatis iaculis. Vivamus viverra purus vel magna. Donec in justo sed odio malesuada dapibus. Nunc ultrices aliquam nunc. Vivamus facilisis pellentesque velit. Nulla nunc velit, vulputate dapibus, vulputate id, mattis ac, justo. Nam mattis elit dapibus purus. Quisque enim risus, congue non, elementum ut, mattis quis, sem. Quisque elit.