ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

Scuola di Ingegneria e Architettura
Dipartimento di Informatica - Scienza e Ingegneria - DISI
Corso di Laurea Magistrale in Ingegneria Informatica

PROGETTAZIONE E IMPLEMENTAZIONE DI UN CHATBOT INTELLIGENTE TRAMITE PIATTAFORMA LANGCHAIN: STUDIO E VALUTAZIONE DEI VECTORDB

Relatore:
Prof. Matteo Poggi

Presentata da:
Michelangelo Florio

Anno Accademico 2023/2024

"Computing is not about computers any more. It is about living."

— Nicholas Negroponte.

Abstract

In questo elaborato di tesi sarà trattato il progetto che consiste nella progettazione e nello sviluppo di un chatbot basato su un RAG, implementato mediante l'utilizzo della piattaforma Langchain, per conto dell'azienda Data Reply srl. Innanzitutto, saranno presentati argomenti come Generative AI, LLM e LangChain, sarà fornita una definizione del concetto di RAG, sarà esplicitata la pipeline mediante la quale si è giunti al chatbot finale e saranno descritte nel dettaglio le API utilizzate. In secondo luogo, sarà trattata la tematica relativa ai vectorDB, analizzandone la natura, il funzionamento e le prestazioni derivanti da ognuno di essi.

Indice

Prefazione **11**

1 Introduzione **13**

1.1 Generative AI 13

1.2 LLM 14

1.3 LangChain 15

2 RAG **19**

2.1 Pipeline 19

2.1.1 Loading 19

2.1.2 Chunking 20

2.1.3 Vectorization 20

2.1.4 Prompting 20

3 Implementazione software **23**

3.1 LangChain API 23

3.1.1 Loaders 24

3.1.2 Text splitters 25

3.1.3 Embeddings 26

3.1.4 VectorDB 26

3.1.5 Prompt templates 28

3.1.6 Similarity search 33

3.2 Tecniche di search 34

3.3 Risoluzione dei pain points 39

3.3.1 Cleaning with unstructured.io 40

3.3.2 Contextual Compression 41

3.3.3 Long Context Reorder 42
3.3.4 Reranking 43

4 Prototipo **45**
4.1 Caso di studio 45
4.2 Requisiti 46
4.3 Demo 47

5 Risultati sperimentali **53**
5.1 VectorDB e tecniche di search 53
5.2 Chunks 58
5.3 Soluzioni adottate per i pain points 59

Conclusioni **65**

Ringraziamenti **67**

Elenco delle figure

1.1 Impatto economico della generative AI sull'economia globale ($ trillion) . . 13
1.2 Panoramica di una pipeline implementata mediante LangChain 17
2.1 Figura che rappresenta i vari stadi della pipeline implementata [8] 21
3.1 Confronto in termini di accuracy tra Zero Shot APE e Zero Shot [12] . . . 32
3.2 Panoramica sulle principali tecniche di prompting [25] 33
3.3 Hybrid Search Pinecone 36
3.4 Esempio di pipeline in cui viene impiegata la libreria unstructured.io . . . 40
3.5 Esempio di pipeline in cui viene impiegata la Contextual Compression . . . 42
3.6 Esempio di pipeline in cui viene impiegato il Long Context Reorder 43
3.7 Esempio di pipeline in cui viene impiegato il Reranking 44
4.1 Versione iniziale del chatbot 47
4.2 Seconda versione del chatbot con sidebar e sliders integrati 48
4.3 Home page della demo realizzata per l'evento 49
4.4 Pagina dedicata all'ambito banking 50
4.5 Pagina dedicata all'ambito insurance 51
5.1 Esempio di pipeline in cui viene integrata l'evaluation con Ragas [28] . . . 53
5.2 Score calcolati sulla Keyword e Semantic Search del Chroma vectorDB . . 55
5.3 Score calcolati sulla Keyword e Semantic Search del FAISS vectorDB . . . 56
5.4 Score calcolati sulle tecniche di search implementate per Chroma, Databricks e Pinecone vectorDB 57
5.5 Score calcolati utilizzando diverse chunk_size 58
5.6 Score calcolati utilizzando diverse chunk_overlap 59
5.7 Score calcolati in seguito all'applicazione del cleaning con unstructured.io . 59

5.8 Analisi sull'impiego della Contextual Compression 61
5.9 Analisi sull'impiego del Long Context Reorder 62
5.10 Analisi sull'impiego del Reranking con FlashrankRerank 63
5.11 Analisi sull'impiego del Reranking con Ranker 63

Prefazione

La Generative AI, conosciuta anche come Intelligenza Artificiale Generativa, è una specializzazione dell'Intelligenza Artificiale tradizionale che si concentra sulla creazione di nuovi contenuti di diverso tipo, che vengono generati artificialmente da un modello LLM. Proprio questo, è il dettaglio che differenzia la Generative AI dall'AI tradizionale, in quanto la creazione dei contenuti avviene servendosi unicamente dei dati di addestramento in possesso del modello. Negli ultimi anni il mondo della Generative AI è stato protagonista di una crescita esponenziale, che senza dubbio non si arresterà in futuro. Si prevede infatti che, nei prossimi anni, essa contribuirà in modo significativo all'espansione dell'economia globale.

I Large Language Models (LLM) sono la tecnologia fulcro della Generative AI. Essi non sono altro che modelli di Intelligenza Artificiale che vengono preaddestrati su dataset molto grandi e sono in grado di comprendere e generare il linguaggio umano. Per fare ciò, utilizzano delle rappresentazioni numeriche delle parole, che vengono memorizzate all'interno di spazi vettoriali multidimensionali e inserite all'interno di database vettoriali. In questo modo, un LLM riesce a comprendere il significato di parole e frasi e le relazioni semantiche presenti al loro interno.

Uno dei framework più utilizzati per la realizzazione di applicazioni di Generative AI basate su LLM, è LangChain. LangChain è un framework open-source lanciato nel 2022 che offre moltissime API, disponibili in librerie basate su Python e JavaScript, che semplificano notevolmente il processo di sviluppo di chatbot e agenti virtuali. La forza di LangChain è rappresentata principalmente dalla sua modularità, che consente la concatenazione di funzioni e classi di oggetti, riducendo al minimo sia la quantità di codice che il livello di conoscenza richiesta, garantendo così un alto livello di astrazione.

Un RAG (Retrieval-Augmented Generation) è una tecnica che consente il recupero delle informazioni e, a partire da esse, la generazione della risposta. La sua realizzazione

avviene grazie all'implementazione di una pipeline formata da un insieme di stadi, che vanno dal caricamento dei documenti alla generazione della risposta.
Questo elaborato si propone di trattare in modo approfondito il progetto implementato per l'azienda Data Reply srl, che prevede la realizzazione di un chatbot in grado di analizzare un dataset di files PDF e, sulla base delle informazioni acquisite, rispondere alle query fornite in input dall'utente. Sarà analizzato il progetto nella sua interezza, dalle API di LangChain impiegate, alle demo realizzate, ponendo particolare attenzione sui vectorDB. A tal proposito, essi saranno trattati in modo completo, a partire dalla definizione di database vettoriale, fino a una descrizione esauriente dei dettagli implementativi. Saranno esaminate a fondo le tecniche di search applicate, analizzando la natura e la struttura di supporto che i vari vectorDB offrono in tal senso. Saranno presentate delle soluzioni relative ad alcuni pain points comuni ai RAG, riportandone i benefici acquisiti dal punto di vista prestazionale. Infine, saranno presi in esame i risultati ottenuti da alcuni test effettuati, al fine di poter mettere in luce le performance di ciascun vectorDB impiegato durante l'implementazione del progetto.

Organizzazione

Il seguente elaborato è suddiviso in 5 parti.
Nel primo capitolo sarà introdotta la Generative AI, gli LLM e sarà fornita una panoramica sul framework LangChain, largamente utilizzato nello sviluppo del progetto descritto in questo documento.
Nel secondo capitolo sarà illustrato il concetto di RAG e sarà descritta la pipeline su cui si basa, presentando una breve spiegazione per ogni stadio da cui essa è costituita.
Nel terzo capitolo sarà fornito un background relativo alla documentazione e saranno trattate nel dettaglio le API utilizzate.
Nel quarto capitolo, sarà delineato il caso di studio che ha portato alla realizzazione del progetto, saranno esplicitati i requisiti e saranno presentati i prototipi implementati.
Infine, nel quinto capitolo, saranno analizzati e discussi i risultati ottenuti da alcuni test effettuati.

Capitolo 1

Introduzione

1.1 Generative AI

La Generative AI o Intelligenza Artificiale Generativa, è una forma relativamente innovativa di Intelligenza Artificiale che, a differenza dei suoi predecessori, consente di creare dei contenuti nuovi servendosi unicamente dei dati di addestramento di cui è in possesso. La capacità di produrre testi, immagini, audio e video simili a quelli prodotti dall'uomo, ha fatto in modo che la sua popolarità crescesse in modo esponenziale, sin da quando il primo chatbot di AI generativa fu rilasciato da OpenAI, nel novembre 2022. Per comprendere meglio la rapida ascesa di cui la genAI è stata protagonista negli ultimi anni, basti pensare che un report di McKinsey & Company risalente a giugno 2023 ha stimato che il suo impiego potrebbe potenzialmente contribuire ad un aumento dell'economia globale che va tra i 6100 e i 7900 miliardi di dollari all'anno [2].

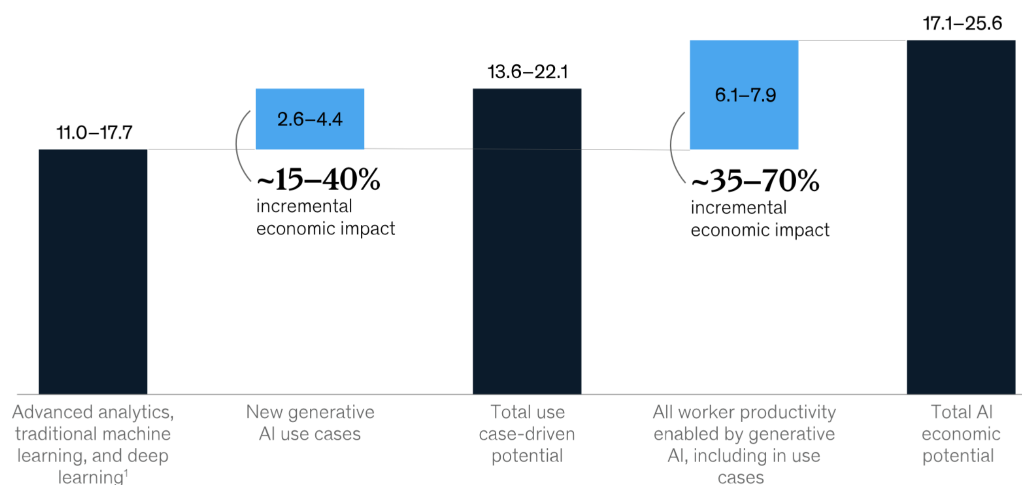

Figura 1.1: Impatto economico della generative AI sull'economia globale ($ trillion)

Finora è stato possibile comprendere l'importanza della Generative AI e come essa rappresenti una sorta di "sottocategoria" dell'Intelligenza Artificiale, tuttavia è doveroso spiegare che cos'è e come funziona nella pratica. Per genAI si intendono quelle tecnologie che, mediante l'utilizzo di modelli di Machine Learning, spesso basati su reti neurali profonde, riescono a generare risposte e dati di varie tipologie utilizzando unicamente il dataset di addestramento originale e un prompt testuale fornito dall'utente.
È necessario in ogni caso precisare che, congiuntamente alle sue notevoli prospettive di produttività, la Generative AI porta con sè nuovi potenziali rischi, sia a livello aziendale che privato. Alcune criticità sono inevitabilmente legate alla privacy, alla capacità di provocare interruzioni economiche su larga scala e alle "allucinazioni" dei modelli di ML. Malgrado ciò, dando uno sguardo al futuro, è innegabile che il suo impiego possa apportare numerosi benefici sia in campo economico che sociale [1].

1.2 LLM

I Large Language Models sono modelli di Intelligenza Artificiale preaddestrati su enormi quantità di dati, che utilizzano tecniche di Machine Learning e di Natural Language Processing (NLP) e che consentono la comprensione e la generazione di linguaggio umano. Nello specifico, gli LLM utilizzano i cosiddetti Trasformers, insiemi di reti neurali costituite ciascuna da un encoder e da un decoder, che permettono di estrarre e riconoscere la grammatica, i significati e l'intento di una sequenza di testo, al fine di comprendere le relazioni tra parole e frasi in essa contenute.
L'architettura delle reti neurali, che gli LLM utilizzano, porta ad avere modelli molto grandi, spesso composti da centinaia di miliardi di parametri, come ad esempio GPT-3, che ne vanta ben 175 miliardi.
Un fattore chiave per quanto riguarda il funzionamento degli LLM è rappresentato proprio dal modo in cui rappresentano le parole. Le forme precedenti di Machine Learning utilizzavano una tabella numerica per rappresentare ogni singola parola, il che non consentiva di riconoscere le relazioni tra di esse, ad esempio quelle con significati simili. La svolta mediante la quale gli LLM sono riusciti ad aggirare tale problema risiede proprio in una nuova tipologia di rappresentazione, che consiste nell'impiego di

vettori multidimensionali, denominati embeddings. In questo modo, parole con significati contestuali affini o con altre tipologie di relazioni, sono vicine tra loro nello spazio vettoriale, rendendo possibile una piena conoscenza del linguaggio da parte dei modelli.
Le applicazioni degli LLM sono molteplici, dalla generazione di frasi all'analisi del sentiment di un testo, dall'estrazione di informazioni da documenti di grandi dimensioni all'implementazione di chatbot interattivi [4].

1.3 LangChain

LangChain è un framework open-source lanciato da Harrison Chase nell'ottobre 2022, che viene utilizzato per la creazione di applicazioni basate sui Large Language Models. Tale framework offre svariate API, disponibili in librerie basate su Python e JavaScript, che semplificano il processo di creazione di applicazioni basate su LLM, come chatbot e agenti virtuali.
L'importanza di LangChain è dovuta a molteplici fattori. Innanzitutto, attraverso il suo impiego, è possibile riutilizzare gli LLM per applicazioni specifiche, senza dover di nuovo effettuare la fase di addestramento. In secondo luogo, esso semplifica lo sviluppo dell'Intelligenza Artificiale attraverso l'astrazione, semplificando il codice mediante la rappresentazione di uno o più processi complessi sotto forma di un'unica componente. Ultimo, ma non per importanza, tale framework è open-source ed è supportato da una vasta comunità di sviluppatori molto attiva [6].
Il funzionamento di LangChain si basa sull'utilizzo di componenti modulari, come funzioni e classi di oggetti, che fungono da elementi costitutivi e possono essere concatenati insieme per realizzare applicazioni di Generative AI, riducendo al minimo sia la quantità di codice, sia la conoscenza approfondita che attività complesse di NLP richiederebbero. Sebbene l'approccio basato sull'astrazione della piattaforma da un lato possa parzialmente limitare un programmatore esperto nella personalizzazione di una specifica applicazione, dall'altro consente sia a specialisti che principianti la creazione di prototipi in modo molto rapido e accessibile.
Le risorse che LangChain mette a disposizione sono le seguenti:

- **LLM**: come in parte anticipato, è possibile importare ed utilizzare molti modelli

open-source come LLaMa o Hugging Face; inoltre, se si dispone di un'API key, è possibile accedere anche a modelli proprietari, come quelli offerti da OpenAI.

- **Prompt templates**: sono messe a disposizione delle classi che formalizzano la composizione dei prompt, ovvero le istruzioni fornite in input a un modello, in modo da rendere non necessaria la codifica manuale del contesto e delle query.
- **Chains**: come suggerisce il nome, sono il fulcro dei workflow di LangChain e consentono l'unione degli LLM ad altre componenti.
- **Indexes**: con questo termine ci si riferisce a tutta la documentazione esterna a cui i modelli devono accedere per svolgere determinate funzionalità, che comprende:
  - **Document Loaders**: consentono il caricamento dei documenti.
  - **Vector databases**: memorizzano delle rappresentazioni numeriche sotto forma di vettori.
  - **Text splitters**: vengono impiegati nella suddivisione di documenti testuali di grandi dimensioni in parti più piccole.
  - **Retrieval**: permette di recuperare e integrare rapidamente le informazioni rilevanti.
- **Memory**: sono delle utility che vengono utilizzate per conservare tutte le conversazioni o parti di esse, al fine di aggiungere memoria a un sistema.
- **Agents**: sono degli strumenti che possono utilizzare un determinato modello come motore di ragionamento per stabilire le azioni da intraprendere; una chain creata per un agent solitamente comprende un elenco dei tool disponibili, l'input dell'utente e tutte le fasi pertinenti eseguite in precedenza.
- **Tools**: sono un insieme di funzioni che consente agli agents di interagire con le informazioni disponibili nel mondo reale, al fine di espandere o migliorare i servizi che essi sono in grado di fornire; Wolfram Alpha, Wikipedia e Google Search sono alcuni esempi di tools che LangChain offre [5].

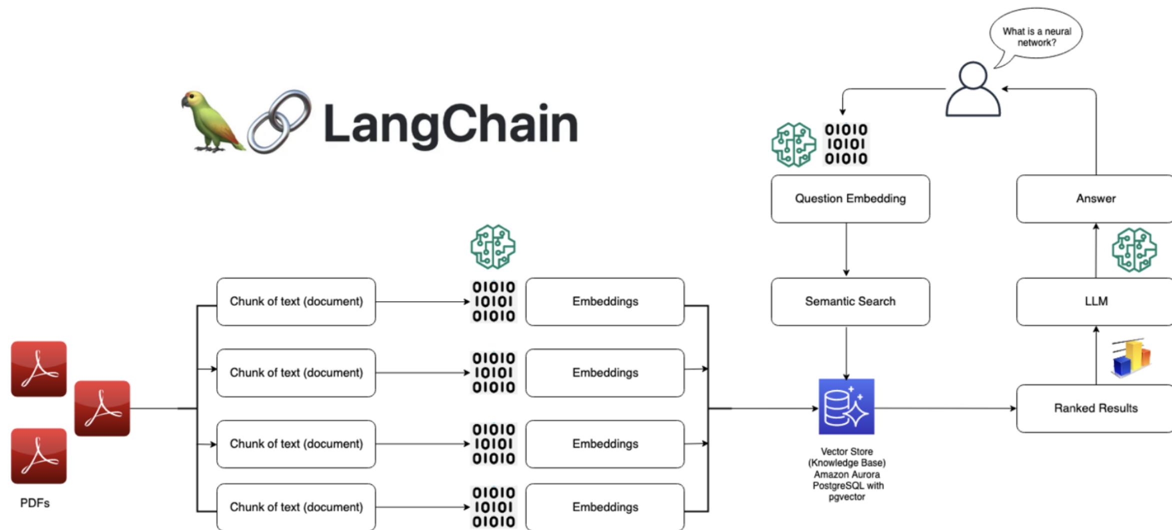

Figura 1.2: Panoramica di una pipeline implementata mediante LangChain

Capitolo 2

RAG

La progettazione del chatbot trattato in questo elaborato utilizza come punto di partenza l'implementazione di un RAG. Un RAG (Retrieval-Augmented Generation) è una metodologia che combina tecniche di recupero delle informazioni (Retrieval) con tecniche di generazione del linguaggio naturale (Generation). Nello specifico, i metodi di retrieval hanno come obiettivo quello di andare a ricercare e selezionare oggetti *Document* rilevanti attingendo alle informazioni contenute nel dataset, in modo da poter rispondere a una determinata query. D'altro canto, le procedure di generazione della risposta sono affidate all'LLM, che, sulla base delle informazioni di contesto recuperate dal retriever e della query dell'utente, genera la risposta [10].

2.1 Pipeline

Lo sviluppo del RAG è stato progressivamente realizzato mediante l'implementazione di una pipeline composta da una serie di stadi successivi, descritti nei paragrafi seguenti.

2.1.1 Loading

Nella fase di loading viene condotta un'operazione cruciale volta al caricamento dei files contenuti nel dataset e alla trasformazione del contenuto delle pagine in una lista di oggetti *Document*, che vengono poi utilizzati nelle fasi successive. Per quanto riguarda questo step, ci sono degli accorgimenti da adottare a seconda delle casistiche. Ovviamente vi è una fase preliminare, che consiste nel decidere quali formati di documenti andare a gestire, così da utilizzare dei loader specifici e ottimizzati, che spesso risultano più efficienti.

Dopodichè, è doveroso verificare la qualità del dataset, fattore da non trascurare ai fini del funzionamento e delle performance del prodotto finale. Inoltre, nel caso di dataset di grandi dimensioni, potrebbe essere utile gestire in modo concorrente l'operazione di caricamento, così da ottimizzare i tempi di esecuzione.

2.1.2 Chunking

Nello stadio di chunking vengono creati i cosiddetti chunks, dei frammenti di testo derivanti dai *Documents* precedentemente caricati. Essi vengono realizzati mediante la definizione di diverse caratteristiche, come la dimensione dei singoli chunk e la sovrapposizione tra questi ultimi. I chunks possono essere costituiti da parole, gruppi di parole o frasi che costituiscono un'unità di significato coeso all'interno del testo. Questa fase è molto importante per l'analisi e l'interpretazione delle relazioni semantiche che vi sono all'interno di un frammento testuale.

2.1.3 Vectorization

Nella fase di vectorization vengono innanzitutto generati gli embeddings, delle rappresentazioni numeriche di parole o frasi all'interno di uno spazio vettoriale, che consentono di catturare i legami semantici presenti. Tale conversione avviene per mezzo di modelli preaddestrati che si occupano di ciò. In secondo luogo, essi vengono caricati all'interno dei vectorDB. I vectorDB sono dei database vettoriali, ottimizzati per permettere un accesso rapido ed efficiente ai dati in essi contenuti. Consentono di memorizzare, indicizzare e recuperare all'occorrenza gli embeddings. Grazie alle loro performance in termini di memorizzazione e tempi di accesso, agevolano la gestione di grandi volumi di embeddings.

2.1.4 Prompting

In questo ultimo step della pipeline, a partire dal retriever, ottenuto dal vectorDB definito nello stadio precedente, è possibile applicare delle tecniche di search al fine di acquisire i k oggetti *Document* che presentano la più alta cosine similarity relativamente alla query fornita in input dall'utente. Dopo aver ottenuto tali oggetti, è possibile creare la chain, che il modello utilizza per rispondere alla domanda. Esistono diverse tecniche

di prompting, che talvolta agevolano il modello nella generazione della risposta, ma esse verranno trattate in modo più approfondito nel capitolo successivo.

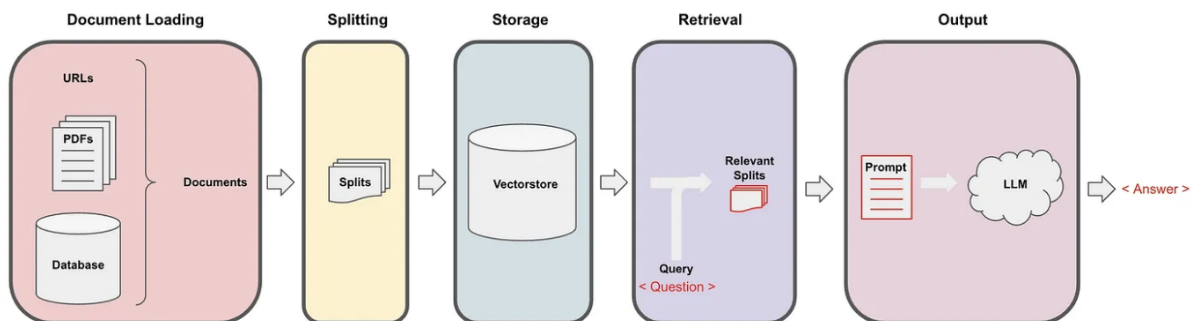

Figura 2.1: Figura che rappresenta i vari stadi della pipeline implementata [8]

Capitolo 3

Implementazione software

Come in parte accennato, la presente tesi tratta di un progetto implementato congiuntamente al collega Manuel Greco, per conto dell'azienda Data Reply srl. Considerata la complessità del progetto, in questo capitolo si è deciso di descrivere in modo esauriente e completo esclusivamente la parte implementativa relativa ai vectorDB, mentre gli aspetti relativi al prompting saranno trattati solo parzialmente, poichè saranno approfonditi in modo altrettanto esauriente nell'elaborato di tesi del collega Manuel Greco. In questo capitolo saranno esaminati i database vettoriali utilizzati, la struttura di supporto che li caratterizza e le tecniche di search che essi forniscono ai fini del retrieval delle informazioni, riportando inoltre alcuni frammenti di codice impiegati durante la fase di implementazione.

3.1 LangChain API

Questa sezione si propone di descrivere in modo dettagliato le varie API di LangChain utilizzate.
È da premettere che, per una questione di pulizia e ordine del codice, esse sono state utilizzate all'interno di funzioni create e contenute in file *.py* appositi, che sono stati raggruppati in base alla tipologia e all'ambito di utilizzo. Tale approccio ha consentito di beneficiare di una buona organizzazione del codice e di agevolare il riutilizzo futuro di tutte le funzioni realizzate.

3.1.1 Loaders

LangChain offre moltissime API che si occupano del loading di documenti. Nello specifico, la libreria *langchain_community.document_loaders* offre molteplici loaders che si differenziano in base alla tipologia dei documenti da caricare. Per quanto concerne il loading implementato nel progetto discusso in questa tesi, in una fase iniziale è stato utilizzato un *DirectoryLoader*, come segue:

```
def load(path):
    loader = DirectoryLoader(path)
    docs = loader.load()
    return docs
```

Come si evince dal codice riportato, tale API non fa altro che effettuare il caricamento dei file a partire da una directory, il cui path viene passato come argomento al metodo *load*, e restituisce una lista di oggetti *Document* che racchiude il contenuto delle varie pagine dei documenti presenti nel dataset. Questa API è generica per una directory e, in quanto tale, non consentiva di avere buone prestazioni a livello di tempistiche. Si è deciso così di optare poi per dei loader più specifici. Dal momento in cui l'esigenza del progetto era quella di effettuare il caricamento unicamente di files PDF, è stato impiegato un *PyPDFDirectoryLoader*:

```
def load(path):
    loader = PyPDFDirectoryLoader(path)
    docs = loader.load()
    return docs
```

Come si può notare, il comportamento è analogo a quello del loader precedente, ad eccezione per il fatto che il *PyPDFDirectoryLoader* richieda una directory contenente unicamente files in formato PDF. Tale scelta ha consentito una notevole riduzione dei tempi di esecuzione, passando dai 6 minuti iniziali a circa 90 secondi, per lo stesso dataset, che contava 106 files. Sono stati testati anche due ulteriori loaders, *PyPDFLoader* e *PDFReader*, che consentono il loading di singoli file e non di intere directory, tuttavia, in fase di implementazione, si è optato per l'utilizzo di *PyPDFDirectoryLoader*, che offre buone prestazioni e necessita di una singola chiamata all'API.

3.1.2 Text splitters

Per la fase di chunking si è deciso di utilizzare degli "splitters by character", che suddividono un blocco testuale in base a determinati caratteri. Relativamente a questa tipologia, LangChain offre il *RecursiveCharacterTextSplitter* e il *CharacterTextSplitter*. Dal momento in cui i due text splitters si comportano in maniera analoga, è stato utilizzato il *RecursiveCharacterTextSplitter*, come mostrato in seguito:

```
def split(docs, chunk_size, chunk_overlap):
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size = chunk_size,
        chunk_overlap = chunk_overlap
    )
    splits = text_splitter.split_documents(docs)
    return splits
```

Dal codice riportato emerge come innanzitutto sia necessario passare come argomenti alla funzione la lista dei documenti precedentemente caricati, la *chunk_size* e il *chunk_overlap*. Come in parte accennato nel capitolo precedente, la *chunk_size* indica la dimensione dei singoli chunk o blocchi che vengono generati, mentre il *chunk_overlap* definisce la sovrapposizione di questi ultimi. Il comportamento della funzione *split* riportata è molto semplice: bisogna innanzitutto inizializzare lo splitter con i parametri richiesti e poi è possibile invocare il metodo *split_documents*, che restituisce una lista di oggetti *Document* che sono stati suddivisi a partire da un blocco testuale più grande. Il criterio in base al quale avviene tale procedimento, come anticipato, è per carattere, tuttavia, è doveroso sottolineare il fatto che la suddivisione non viene applicata in modo casuale, bensì utilizzando la seguente lista di caratteri di default:

```
["\n\n", "\n", " ", ""]
```

In questo modo, lo splitter cerca di suddividere il testo in corrispondenza di caratteri di fine riga e, quando ciò non è possibile, alla fine di una parola. Tale logica ha ovviamente l'obiettivo di cercare di mantenere invariati i legami sintattici tra le parole presenti all'interno di una frase, ove possibile. Ad ogni modo, l'API consente comunque l'utilizzo di altre stringhe o caratteri diversi da quelli impiegati a default, passandoli come argomento.

3.1.3 Embeddings

LangChain offre svariate tipologie di embeddings, a seconda del modello che si vuole utilizzare, sia open-source che proprietari. Nell'implementazione del progetto, disponendo di un'API key OpenAI, si è deciso di impiegare l'API *OpenAIEmbeddings*, che a default utilizza il modello *text-embedding-ada-002*. Tale modello genera dunque un vettore costituito da 1536 elementi floating point e la "distanza" tra due vettori misura il loro grado di correlazione. Più precisamente, piccole distanze indicano un alto grado di correlazione, viceversa, grandi distanze denotano un basso grado di correlazione [24].

3.1.4 VectorDB

Relativamente ai vectorDB, la libreria *langchain.vectorstores* ne offre diversi e, come nel caso dei modelli, sia open-source, sia proprietari. Durante la fase di sviluppo sono stati utilizzati fondamentalmente quattro vectorDB, sia a scopo implementativo che di testing:

- **Chroma**
- **FAISS**
- **Pinecone**
- **Databricks**

Di seguito è riportata una delle funzioni utilizzate, in questo caso per la creazione di un Chroma vectorDB, memorizzando al suo interno gli embeddings calcolati.

```
def get_ChromaVectorDB(documents):
    embeddings = OpenAIEmbeddings()
    vectordb = Chroma.from_documents(documents = documents, embedding = embeddings)
    return vectordb
```

In primo luogo, il parametro che viene passato alla funzione è costituito dalla lista di oggetti *Document* ottenuti in seguito allo step di chunking. In secondo luogo, dal codice è possibile comprendere come, a partire dal modulo *Chroma*, precedentemente importato dalla libreria sopracitata, sia possibile creare, in pochissime righe di codice, una struttura vettoriale. Nel dettaglio, si è deciso di utilizzare il metodo *from_documents* che, a partire dai documenti chunkizzati e dagli embeddings calcolati, restituisce un Chroma vectorDB. Inoltre, durante la fase di realizzazione dei vector database Chroma e FAISS, si è deciso

di applicare il pattern Factory di Python. Tale pattern non fa altro che incapsulare la logica di creazione degli oggetti all'interno di una classe Factory, che è responsabile della creazione di oggetti, ma allo stesso tempo non espone direttamente la logica di creazione. Per fare ciò, esso sfrutta un metodo astratto, che viene poi ridefinito all'occorrenza dalle classi concrete, come mostrate in seguito:

```
class ExporterFactory(ABC):
    @abstractmethod
    def get_vectorDB(self) -> any:

class ChromaExporter(ExporterFactory):
    def get_vectorDB(self) -> Chroma:
        return Chroma

class FAISSExporter(ExporterFactory):
    def get_vectorDB(self) -> FAISS:
        return FAISS

def factory_vectorDB(documents) -> ExporterFactory:
    factories = {
        "Chroma": ChromaExporter(),
        "FAISS": FAISSExporter()
    }
    while True:
        export = input("Choose the desired vectorDB (Chroma or FAISS): ")
        if export in factories:
            return factories[export].get_vectorDB().from_documents(documents =
                documents, embedding = OpenAIEmbeddings())
        print(f"Unknown vector store: {export}.")
```

Come si può notare, il metodo *get_vectorDB* è astratto e viene ridefinito dalle classi Exporter di Chroma e FAISS. La funzione *factory_vectorDB* non fa altro che chiedere all'utente quale vector database desidera e restituire la corrispondente classe Exporter, che si occuperà poi di restituire a sua volta il vectorDB richiesto dall'utente. I benefici derivanti dall'impiego del pattern Factory sono molteplici [11]:

- **Incapsulamento della creazione degli oggetti**: la complessità relativa alla creazione degli oggetti viene interamente incapsulata all'interno di una classe Factory dedicata.

- **Astrazione delle implementazioni concrete**: il codice lato client non dipende dalle implementazioni concrete degli oggetti, il che porta ad avere un alto livello di astrazione.
- **Facilità di estensione**: aggiungere implementazioni di nuovi oggetti è molto semplice, in quanto è sufficiente creare una classe che reimplementa tale metodo, come nel caso delle classi Exporter viste in precedenza.

Proprio in relazione all'ultimo punto, è doveroso precisare che tale pattern è stato adottato solamente per Chroma e FAISS, in quanto, per via del fatto che sono entrambi open-source, sono stati ampiamente utilizzati, soprattutto in una fase iniziale. Tuttavia, ciò non toglie che esso possa essere facilmente esteso anche agli altri vectorDB, aggiungendo banalmente una classe Exporter che ridefinisca il metodo astratto *get_vectorDB*.

3.1.5 Prompt templates

Per quanto concerne il prompting, LangChain mette a disposizione una serie di API mediante la libreria *langchain_core.prompts*, come *PromptTemplate* e *ChatPromptTemplate*, che consentono la creazione di un prompt template, essenziale per la generazione della chain. Proprio la chain sarà l'oggetto che verrà invocato per ricevere in output una risposta sulla base di una query fornita dall'utente. Nel progetto trattato, essa è stata ottenuta mediante la seguente funzione:

```
def getChain(retriever):
    template = """Answer the question based only on the following context:
    {context}

    Question: {question}
    """
    prompt = ChatPromptTemplate.from_template(template)
    model = ChatOpenAI(temperature = 0)
    chain = (
        {"context": retriever, "question": RunnablePassthrough()}
        | prompt
        | model
        | StrOutputParser()
    )
    return chain
```

Analizzando il codice, è doveroso precisare innanzitutto che il retriever passato come argomento può essere semplicemente ottenuto a partire dal vectorDB mediante il metodo *as_retriever*, grazie al quale è possibile ottenere un oggetto *VectorStoreRetriever*, come mostrato in seguito:

```
retriever = vectorDB.as_retriever()
```

Tornando alla funzione *getChain*, si può notare come innanzitutto venga creato un template sotto forma di stringa, che poi viene trasformato in un oggetto *ChatPromptTemplate* dal metodo *ChatPromptTemplate.from_template*. In secondo luogo, viene definito il modello: si è scelto di utilizzare l'API *ChatOpenAI*, che a default prevede l'impiego di *gpt-3.5-turbo*. La temperature, che nel caso in questione viene passata al *ChatOpenAI* e settata a 0, è un parametro che caratterizza gli LLM, in particolare va a condizionare la generazione della risposta alla query da parte del modello. Infine, vi è la creazione della chain, che sfrutta il retriever come contesto, un oggetto *RunnablePassthrough* come question, il prompt e il modello, precedentemente definiti, e un oggetto *StrOutputParser*. L'oggetto *RunnablePassthrough* consente di passare l'input senza apportare alterazioni ad esso, mentre l'oggetto *StrOutputParser* si occupa del parsing della risposta del modello sotto forma di stringa.
Per la parte di prompting inerente al chatbot realizzato, sono state adottate fondamentalmente quattro tecniche:

- **Zero Shot**

```
if type == "zero":
template = """Answer the question based only on the following context:
{context}
Question: {question}"""
prompt = ChatPromptTemplate.from_template(template)
model = ChatOpenAI(temperature=0.3)
chain = (
    {"context": retriever, "question": RunnablePassthrough()}
    | prompt
    | model
    | StrOutputParser()
)
```

- **Zero Shot APE**

```
elif type == "zeroAPE":
template = """Answer the question based only on the following context:
{context}
Question: {question}
Let’s work this out in a step by step way to be sure we have the right
    answer."""
prompt = ChatPromptTemplate.from_template(template)
model = ChatOpenAI(temperature=0.3)
chain = (
    {"context": retriever, "question": RunnablePassthrough()}
    | prompt
    | model
    | StrOutputParser()
)
```

- **One Shot**

```
elif type == "one":
template = """Answer the question based only on the following context:
{context}
For example: day 1 - beginning balance: 10, new advances: 5, payments or
    credits: 4.
            day 2 - beginning balance: 11, new advances: 6, payments or
                credits: 1.
            day 3 - beginning balance: 16, new advances: 1, payments or
                credits: 0.
            For each day you have to calculate the daily balance as:
            beginning balance + new advances - payments or credits (10 + 5
                -4) = 11
            Repeat this operation for each day.
            Calculate the average daily balance as the ratio between the sum
                of the
            daily balances and the number of the days:
            (11 + 16 + 17) = 44
            44 / 3 = 14.67
            Calculate carefully the final ratio.
Question: {question}"""
prompt = ChatPromptTemplate.from_template(template)
model = ChatOpenAI(temperature=0)
chain = (
```

```
    {"context": retriever, "question": RunnablePassthrough()}
    | prompt
    | model
    | StrOutputParser()
)
```

- **Few Shot**

```
elif type == "few":
template = """Answer the question based only on the following context:
{context}
example 1: day 1 - beginning balance: 10, new advances: 5, payments or
    credits: 4.
           day 2 - beginning balance: 11, new advances: 6, payments or
               credits: 1.
           day 3 - beginning balance: 16, new advances: 1, payments or
               credits: 0.
           For each day you have to calculate the daily balance as:
           beginning balance + new advances - payments or credits (10 + 5
               -4) = 11
           Repeat this operation for each day.
           Calculate the average daily balance as the ratio between the sum
               of the
           daily balances and the number of the days:
           (11 + 16 + 17) = 44
           44 / 3 = 14.67
           Calculate carefully the final ratio.
example 2: day 1 - beginning balance: 20, new advances: 5, payments or
    credits: 4.
           day 2 - beginning balance: 21, new advances: 6, payments or
               credits: 1.
           day 3 - beginning balance: 26, new advances: 1, payments or
               credits: 0.
           For each day you have to calculate the daily balance as:
           beginning balance + new advances - payments or credits (20 + 5
               -4) = 21
           Repeat this operation for each day.
           Calculate the average daily balance as the ratio between the sum
               of the
           daily balances and the number of the days:
```

```
            (21 + 26 + 27) = 74
            74 / 3 = 24.67
            Calculate carefully the final ratio.
Suggestion: Let's think step by step
Question: {question}"""
prompt = ChatPromptTemplate.from_template(template)
model = ChatOpenAI(temperature=0)
chain = (
    {"context": retriever, "question": RunnablePassthrough()}
    | prompt
    | model
    | StrOutputParser()
)
```

Come si può dedurre dal codice precedentemente riportato, la differenza tra le tecniche citate risiede proprio nel numero di esempi di addestramento che vengono forniti al modello, in quanto la Zero Shot disporrà unicamente del contesto fornito dal retriever e della query, la One Shot utilizzerà anche un esempio di addestramento e la Few Shot sfrutterà più esempi, che, nel caso del progetto trattato, sono due. La tecnica Zero Shot APE, dove APE sta per Automatic Prompt Engineer, fu proposta in un paper scientifico nel 2022 [13] e differisce dalla Zero Shot dal momento che si concentra sul fornire un prompt più completo e guidato possibile, in modo che l'accuratezza della risposta fornita dal modello sia più alta [12].

No.	Category	Zero-shot CoT Trigger Prompt	Accuracy
1	APE	Let's work this out in a step by step way to be sure we have the right answer.	**82.0**
2	Human-Designed	Let's think step by step. (*1)	78.7
3		First, (*2)	77.3
4		Let's think about this logically.	74.5
5		Let's solve this problem by splitting it into steps. (*3)	72.2
6		Let's be realistic and think step by step.	70.8
7		Let's think like a detective step by step.	70.3
8		Let's think	57.5
9		Before we dive into the answer,	55.7
10		The answer is after the proof.	45.7
-		(Zero-shot)	17.7

Figura 3.1: Confronto in termini di accuracy tra Zero Shot APE e Zero Shot [12]

A tal proposito, analizzando il codice soprastante relativo alla Zero Shot APE, si evince come nel template sia stata aggiunta la frase "Let's work this out in a step by step way

to be sure we have the right answer.", che consente al modello di "ragionare" in modo migliore e restituire risposte più precise.

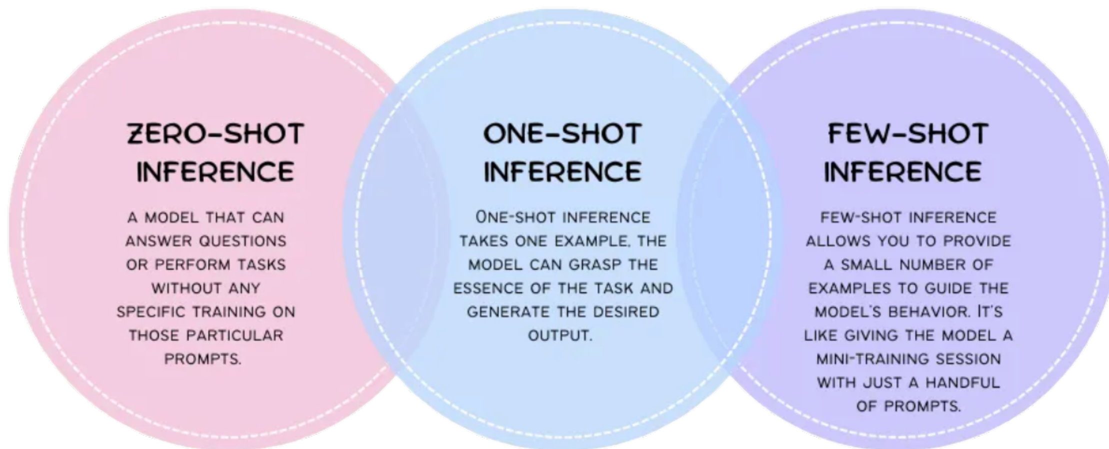

Figura 3.2: Panoramica sulle principali tecniche di prompting [25]

Anche per le tecniche di prompting è stato adottato il pattern Factory, procedendo in modo analogo a quanto fatto per i vectorDB, così da avere una classe Factory che racchiudesse la logica di creazione delle rispettive chain, per tutte le quattro casistiche sopracitate.

Per quanto concerne la generazione della risposta, successiva alla creazione dell'oggetto chain, è stato necessario invocare il metodo *invoke* di cui esso dispone:

```
response = chain.invoke(query)
```

In questo modo è possibile interrogare la chain passando come parametro la query dell'utente ed ottenere in output la rispettiva response.

3.1.6 Similarity search

LangChain mette a disposizione degli oggetti vectorDB come *Chroma* e *FAISS* una serie di metodi che consentono di usufruire di diverse tipologie di similarity search, sia sincrone che asincrone. Nel progetto si è deciso di utilizzare il metodo *similarity_search* come segue:

```
    documents_similarity_search = vectordb.similarity_search(query, k = 10)
```

Nel dettaglio, esso consente di recuperare i k documenti più rilevanti relativamente alla query passata come parametro. La similarità degli oggetti *Document* contenuti nel vectorDB è determinata da alcuni score che vengono calcolati dal metodo *similarity_search_with_score*, invocato dalla *similarity_search* stessa. Tali score sono dei

float che vanno a definire la cosine distance tra ogni documento e la query passata come parametro, che è inversamente proporzionale alla similarità. Di conseguenza score bassi indicano una più alta similarità e viceversa. Di seguito è mostrata la formula che definisce il modello matematico descritto in precedenza [9].

$$similarity(A, B) = cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}$$

Sulla base degli score ottenuti, vengono restituiti i k documenti più simili, sotto forma di *List[Document]*. Un dettaglio che è necessario precisare è che, a default, k è posto uguale a quattro, di conseguenza il metodo *similarity_search* restituirà una lista con i quattro *Document* più simili. Ad ogni modo, è possibile passare il valore di k come parametro a tale metodo, come si può notare nel codice riportato, in modo da avere una lista contenente il numero di documenti desiderato.
Lo stesso risultato della *similarity_search* può essere ottenuto mediante il metodo *get_relevant_documents* invocato sul retriever. Tuttavia, siccome in alcune versioni recenti di LangChain esso è deprecato, è consigliato l'utilizzo del metodo *invoke* al suo posto, che presenta un comportamento analogo.

3.2 Tecniche di search

Per quanto concerne il retrieval delle informazioni a partire dai vectorDB impiegati, oltre alla *similarity_search* di LangChain, sono stati utilizzati ulteriori approcci. Per fini analitici, l'azienda ha richiesto l'implementazione di tre tecniche di search per ogni database vettoriale, in modo da raccogliere dati utili in merito alle performance di ognuno di essi. Ciò ha consentito di stilare un confronto che mira a fare un bilancio relativo alle prestazioni, che però verrà mostrato nella sezione 5.1.
Le tecniche di search sviluppate sono tre:

- **Keyword Search**
- **Hybrid Search**
- **Vector Search (o Semantic Search)**

La *Keyword Search* si basa sull'utilizzo e sull'identificazione di parole o gruppi di parole specifiche, le cosiddette keyword, che consentono il recupero delle informazioni dal

vectorDB, mediante la selezione dei documenti più rilevanti in relazione alla parola chiave fornita. Tale tecnica presenta numerosi benefici soprattutto quando in un documento sono presenti dei termini "esclusivi", ovvero non comuni, che si ripetono poche volte all'interno del blocco testuale. Viceversa, nel momento in cui la keyword fosse utilizzata spesso all'interno del documento, le sue prestazioni potrebbero essere limitate, in quanto si basa unicamente sulla parola chiave, senza sfruttare il significato dell'intera query mediante un embedding di quest'ultima, come fanno altre tecniche.
La *Vector Search* invece impiega un embedding della query per recuperare i documenti più rilevanti all'interno del database vettoriale. In questo modo, tale tecnica consente di basarsi sul significato dell'intera query fornita in input e di usufruire dunque di una sorta di "contesto" rispetto alla *Keyword Search*. Ciò permette generalmente a questo approccio di garantire buone prestazioni.
Infine, la *Hybrid Search*, come si può intuire dal nome, è una tecnica ibrida, che sfrutta i benefici di entrambe quelle sopracitate. Nello specifico, essa sfrutta sia la keyword, che l'embedding della query. Tale strategia si basa sul principio "take the best from both worlds", conducendo spesso a buone performance [17]. Di seguito sono state riportate le tre tecniche di search descritte finora, implementate per i vectorDB Chroma e Pinecone. Non verranno presentate le medesime realizzazioni per gli altri database vettoriali utilizzati, in quanto sono molto simili a quella adottata per Chroma.
Pinecone offre supporto per vettori che contengono sia valori sparsi che densi, il che consente l'impiego dell'*Hybrid Search* su un dato index in modo nativo. I vettori sparsi-densi combinano embeddings sparsi e densi sotto forma di un singolo vettore, consentono di rappresentare diverse tipologie di informazioni ed eseguire diverse search. Il vettore denso è il tipo base di vettore in Pinecone e consente l'applicazione della *Vector Search*, poichè esso non è altro che una struttura vettoriale contenente embeddings, ovvero rappresentazioni numeriche del significato semantico. D'altro canto, i vettori sparsi hanno un numero di dimensioni molto elevato, dove solo una piccola porzione di valori sono diversi da 0. Quando sono impiegati nella *Keyword Search*, ogni vettore sparso rappresenta un documento, le dimensioni rappresentano le parole da un dizionario e i valori segnalano l'importanza di tali parole nel documento. Alcuni algoritmi, come l'algoritmo BM25, calcolano la rilevanza dei documenti di testo in base a una serie di fattori, come il numero di corrispondenze delle keyword e la loro frequenza

nel PDF [18].

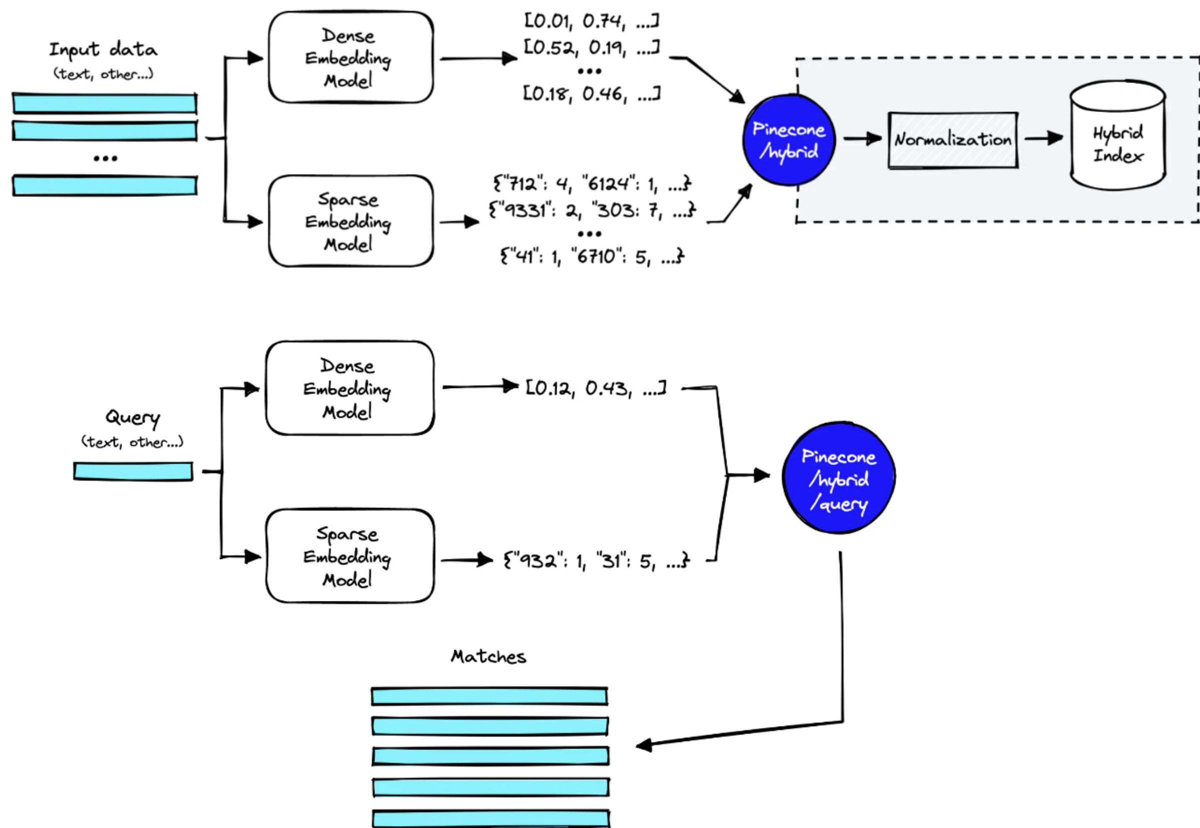

Figura 3.3: Hybrid Search Pinecone

Come anticipato, i vettori sparsi relativi a una determinata query possono essere generati a partire da un encoder come il BM25, come segue:

```
bm25 = BM25Encoder()
doc_sparse_vector = bm25.encode_documents(query)
```

Le tecniche di search implementate nel progetto relativamente al Pinecone vectorDB sono le seguenti:

```
def vector_search(self, vector, top_k):
    results = self.client.query(
        vector = vector,
        top_k = top_k,
        include_values = False,
        include_metadata = True
        )
    return results
```

```
    def hybrid_search(self, query_text, vector, top_k):
        results = self.client.query(
            top_k = top_k,
            vector = vector,
            sparse_vector = query_text,
            include_values = False,
            include_metadata = True
        )
        return results

    def hybrid_search_with_alpha(self, query_text, vector, top_k, alpha):
        hdense, hsparse = hybrid_score_norm(vector, query_text, alpha=alpha)
        results = self.client.query(
            top_k = top_k,
            vector = hdense,
            sparse_vector = hsparse,
            include_values = False,
            include_metadata = True
        )
        return results

    def keyword_search(self, vector, query_text, top_k):
        hdense, hsparse = hybrid_score_norm(vector, query_text, alpha = 0)
        results = query_response = self.client.query(
            top_k = top_k,
            vector = hdense,
            sparse_vector = hsparse,
            include_values = False,
            include_metadata = True
        )
        return results
```

Nella funzione che implementa la *Vector Search* è possibile notare come essa utilizzi il vettore denso, ovvero l'embedding della query fornita in input, per la fase di recupero dei documenti rilevanti. Sono stati riportati due metodi che definiscono l'*Hybrid Search*: uno utilizza direttamente il vettore sparso passato come parametro, mentre l'altro utilizza la funzione *hybrid_score_norm*, che calcola e restituisce i vettori denso e sparso ed opera come segue:

```
def hybrid_score_norm(dense, sparse, alpha: float):
    """Hybrid score using a convex combination

    alpha * dense + (1 - alpha) * sparse

    Args:
        dense: Array of floats representing
        sparse: a dict of ‘indices‘ and ‘values‘
        alpha: scale between 0 and 1
    """
    if alpha < 0 or alpha > 1:
        raise ValueError("Alpha must be between 0 and 1")
    hs = {
        'indices': sparse['indices'],
        'values': [v * (1 - alpha) for v in sparse['values']]
    }
    return [v * alpha for v in dense], hs
```

Come si può osservare, essa calcola il vettore denso sulla base di un valore alpha passato come parametro, che definisce il tipo di search, poichè nella *Hybrid Search* sarà pari a 0.5 e nella *Keyword Search* sarà pari a 0. Infine, la funzione relativa alla *Keyword Search* è molto simile a quella analizzata in precedenza della *Hybrid Search*, ad eccezione per il fatto che in questo caso non verrà utilizzata la funzione *hybrid_score_norm* e il parametro alpha sarà uguale a 0, per eseguire la search sulla base della keyword.
Chroma, rispetto a Pinecone, non offre una struttura di supporto per la *Hybrid Search*, bensì, i suoi vectorDB consentono unicamente di memorizzare gli embedding relativi a una parola o frase. Si è deciso comunque di implementare tale tecnica di search sulla base degli strumenti a disposizione nel seguente modo:

```
def vector_search(self, vector, top_k):
    results = self.client.query(
        query_embeddings = vector,
        n_results = top_k
        )
    return results

def hybrid_search(self, vector, query_text, top_k):
    results = self.client.query(
```

```
            query_embeddings = vector,
            n_results = top_k,
            where_document = {"$contains" : yake.KeywordExtractor(n =
                1).extract_keywords(query_text)[0][0]}
        )
        return results

    def keyword_search(self, query_text, top_k):
        results = self.client.query(
            query_embeddings = generate_embeddings(yake.KeywordExtractor(n =
                1).extract_keywords(query_text)[0][0]),
            n_results = top_k
        )
        return results
```

Come si evince dal codice riportato, la *Vector Search* utilizza unicamente il vettore che rappresenta l'embedding della query, oltre al numero di documenti. La *Keyword Search*, invece che l'embedding della query, utilizza l'embedding della keyword identificata nella query, mediante *yake*, un modulo che consente l'estrazione di parole chiave a partire da una frase. Infine, la *Hybrid Search* impiega entrambi i parametri precedentemente descritti: essa, infatti, innanzitutto utilizza l'embedding della query e, in secondo luogo, tramite il parametro *where_document*, ci si assicura che i document restituiti contengano la keyword in questione, calcolata in modo analogo alla tecnica di search precedentemente trattata.

3.3 Risoluzione dei pain points

Una volta giunti in una fase avanzata di implementazione del RAG, si è deciso di gestire in modo specifico alcuni pain points, prendendo spunto da un articolo che ne tratta i dodici più comuni, offrendo, per ognuno di essi, delle possibili soluzioni [19]. Precisamente, si è deciso di considerare unicamente i pain points più pertinenti allo use case del progetto realizzato. L'adozione di tali accorgimenti ha consentito di migliorare il RAG sviluppato e di ottenere benefici significativi dal punto di vista delle performance, come verrà mostrato nella sezione 5.3. In questa sezione, invece, verranno esaminate le soluzioni adottate, ponendo l'attenzione sull'attuazione pratica.

3.3.1 Cleaning with unstructured.io

La prima soluzione proposta è il *Cleaning dei dati con unstructured.io*, che risolve il pain point relativo a "Missing Content". Una bassa qualità dei dati forniti può compromettere il funzionamento di un RAG, di conseguenza è fondamentale avere a disposizione dei dati "puliti". Alcune strategie che possono essere impiegate per migliorare la qualità del dataset sono:

- Rimozione di informazioni non rilevanti, come caratteri speciali, tag HTML e cosiddette "parole stop", ovvero parole comuni come "il" e "un".
- Identificazione e correzione di errori, come errori di ortografia, refusi ed errori grammaticali.
- Deduplicazione, ovvero rimozione di record duplicati o record simili che potrebbero influenzare il processo di recupero.

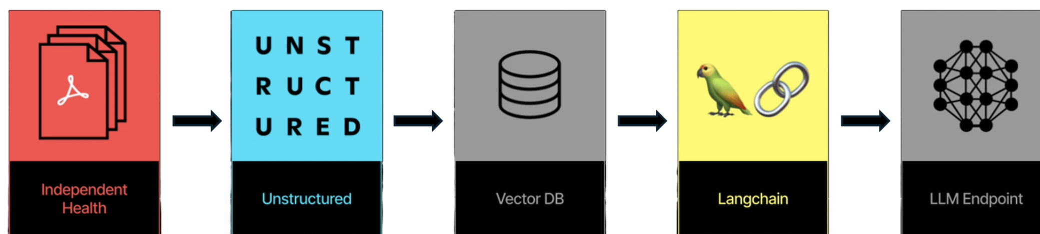

Figura 3.4: Esempio di pipeline in cui viene impiegata la libreria unstructured.io

La libreria *unstructured.io* offre un insieme di funzionalità di pulizia che possono essere utili nell'affrontare tali esigenze [20]. Di seguito verrà riportata la funzione impiegata per eseguire il cleaning descritto:

```
def transformation_pipeline(text):
    text = replace_unicode_quotes(text)
    text = bytes_string_to_string(text)
    text = clean_bullets(text)
    text = clean_dashes(text)
    text = clean_extra_whitespace(text)
    text = clean_non_ascii_chars(text)
    text = clean_ordered_bullets(text)
    text = clean_trailing_punctuation(text)
    return group_broken_paragraphs(text)
```

Come si può notare, la pulizia del *chunk_text* passato come parametro avviene per mezzo di una serie di moduli in pipeline, che la libreria *unstructured.io* offre. Nello specifico, essi vanno a sostituire le virgolette unicode con delle virgolette normali, convertono una stringa di byte in una stringa di testo, raggruppano paragrafi spezzati e rimuovono elenchi puntati, trattini, spazi bianchi extra, caratteri non ASCII, elenchi ordinati e punteggiatura finale. Tale funzione è stata applicata alla colonna del dataframe contenente i *chunk_text*:

```
for index, row in dataset.iterrows():
    dataset.loc[index, "chunk_text"] = transformation_pipeline(dataset.loc[index,
        "chunk_text"])
```

Mediante l'approccio descritto, sono stati ottenuti dei chunks "puliti", che, come sarà mostrato nella sezione relativa ai risultati, ha contribuito ad un miglioramento delle prestazioni del RAG, soprattutto nel caso della *Vector Search*. Essa infatti, basandosi unicamente sul contesto e sul significato della query attraverso l'embedding di quest'ultima, beneficia maggiormente della pulizia apportata rispetto alle altre tecniche di search.

3.3.2 Contextual Compression

Il pain point "Not Extracted", alla base dell'impiego della *Contextual Compression*, deriva dalla memorizzazione di informazioni non rilevanti all'interno dei vectorDB. Ciò comporta innanzitutto l'occupazione di spazio che potrebbe essere utilizzato per informazioni rilevanti e, in secondo luogo, potrebbe distrarre il modello dalle informazioni significative in fase di retrieval dei documenti. Una possibile soluzione per il problema descritto consiste nell'impiego di un *DocumentCompressor* sui documenti recuperati. L'idea alla base del suo funzionamento è la seguente: anzichè restituire immediatamente i documenti ottenuti in seguito alla fase di retrieval, è possibile comprimerli utilizzando il contesto della query fornita, in modo da restituire unicamente le informazioni rilevanti. Con "comprimere" si intende sia la compressione dei contenuti in un singolo documento, sia l'operazione di filtraggio di interi documenti. Tutto ciò si è declinato all'utilizzo del codice riportato in seguito:

```
def return_compression_retriever(retriever):
    llm = OpenAI(temperature = 0)
    compressor = LLMChainExtractor.from_llm(llm)
    compression_retriever = ContextualCompressionRetriever(
```

```
        base_compressor = compressor, base_retriever = retriever
    )
    return compression_retriever
```

Come si può osservare, è stato utilizzato un oggetto *LLMChainExtractor* per generare il compressor a partire dall'LLM e un *ContextualCompressionRetriever* per eseguire la compressione del retriever passato come argomento, sfruttando il compressor precedentemente creato.
In sintesi, dunque, l'obiettivo della *Contextual Compression* è quello di passare al modello più informazioni rilevanti possibili, così da ottimizzare la precisione del RAG.

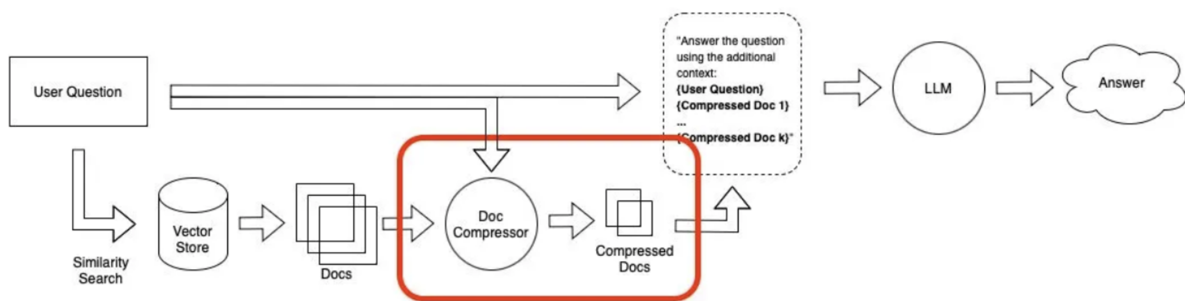

Figura 3.5: Esempio di pipeline in cui viene impiegata la Contextual Compression

3.3.3 Long Context Reorder

Il pain point "Not Extracted" può essere risolto anche mediante l'impiego del *Long Context Reorder*. L'idea nasce da uno studio affrontato in un paper scientifico [22], relativamente all'utilizzo del contesto da parte del modello. In particolare, tale articolo afferma che, sebbene gli LLM siano in grado di gestire lunghi contesti di input, le performance possono degradare significativamente nel momento in cui si cambia la posizione delle informazioni rilevanti. Nello specifico, è stato osservato che le prestazioni sono spesso migliori quando le informazioni rilevanti si trovano all'inizio o alla fine del contesto di input, mentre peggiorano notevolmente quando esse si trovano nel mezzo di contesti lunghi. Ciò denota una criticità dei modelli relativamente all'accesso alle informazioni contenute internamente a contesti molto lunghi, nonostante essi fossero addestrati appositamente a tale scopo.

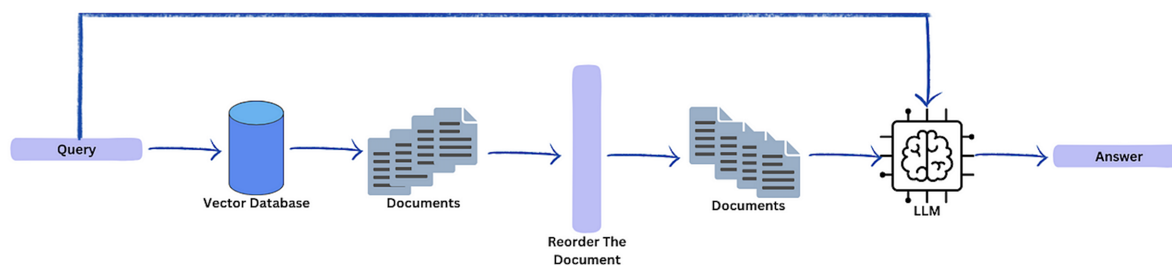

Figura 3.6: Esempio di pipeline in cui viene impiegato il Long Context Reorder

La soluzione proposta per risolvere il problema descritto prevede l'applicazione di un oggetto *LongContextReorder*, reso disponibile dalla libreria *langchain_community.document_transformers*. Esso non fa altro che andare a riordinare i documenti, in modo tale che quelli più rilevanti vengano posti all'inizio o alla fine del contesto. A seguire, è stato riportato il codice sviluppato per l'impiego della soluzione descritta:

```
longContextReorder = LongContextReorder()
reordered_docs = longContextReorder.transform_documents(docs)
```

Come si può notare, è possibile riordinare i documenti in modo molto semplice, invocando il metodo *transform_documents* del *LongContextReorder*, passando come parametro la lista dei documenti su cui effettuare tale operazione.

3.3.4 Reranking

Infine, il *Reranking* è la soluzione proposta per risolvere il pain point "Missed the Top Ranked Documents". Esso deriva dal fatto che, durante la fase di "splitting" dei documenti e durante la conversione di questi ultimi in vettori, c'è una potenziale perdita di informazioni, in quanto i vettori rappresentano i contenuti in un formato numerico compresso e risulta difficile mantenere il contesto in chunk piccoli. Inoltre, considerando unicamente i *top_k* risultati della ricerca vettoriale, vengono potenzialmente trascurate informazioni rilevanti. Un modello di *Reranking* è un tipo di modello che calcola un punteggio per ogni coppia query-documento e, sulla base di ciò, va a riorganizzare i risultati della ricerca vettoriale, garantendo che i risultati più rilevanti siano prioritari nella lista di oggetti *Document* restituita. In sintesi, viene effettuata una riclassificazione dei documenti che poi verranno utilizzati dal modello, in modo da massimizzare la qualità della risposta [23].

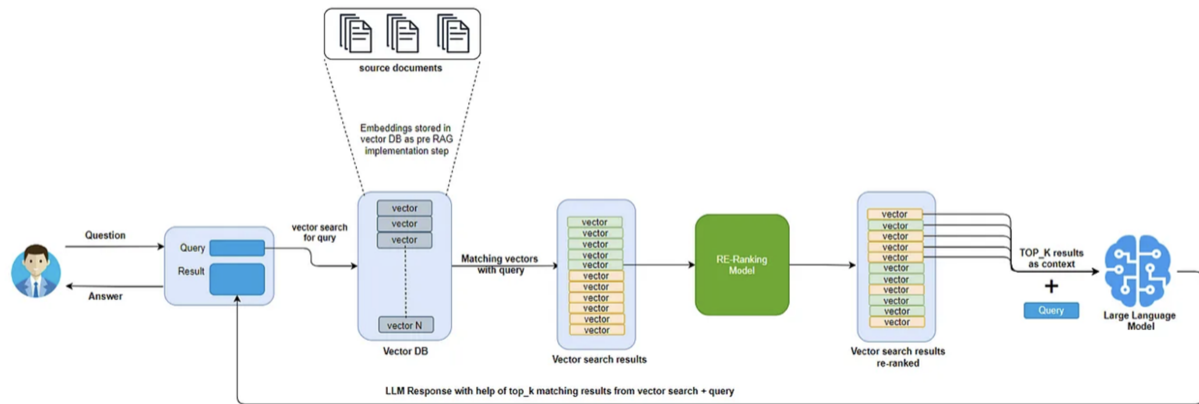

Figura 3.7: Esempio di pipeline in cui viene impiegato il Reranking

Per quanto riguarda l'implementazione della strategia descritta, l'idea iniziale era quella di utilizzare *Cohere*, tuttavia, siccome essa necessita di un'apposita API key, si è deciso di ripiegare sull'impiego del *FlashrankRerank* di LangChain e del *Ranker* di Flashrank:

```
compressor = FlashrankRerank()
compression_retriever = ContextualCompressionRetriever(
    base_compressor = compressor, base_retriever = retriever
)
reranked_docs = compression_retriever.invoke(query)
```

Analizzando il codice, si può notare come il *FlashrankReranker* sfrutti un *ContextualCompressionRetriever* per generare il retriever compresso, che a sua volta porterà ad ottenere i documenti "reranked".

```
docs_ret = retriever.invoke(query)
passages = [
     {"id": i,
      "text": doc.page_content,
      "meta": doc.metadata}
     for i, doc in enumerate(docs_ret)
 ]
ranker = Ranker()
rerankrequest = RerankRequest(query = query, passages = passages)
reranked_docs = ranker.rerank(rerankrequest)
```

Infine, il *Ranker* utilizza una *RerankRequest* che, grazie alla query e ai passages, definiti come mostrato nel codice soprastante, consente la generazione dei documenti riclassificati.

Capitolo 4

Prototipo

4.1 Caso di studio

Come mostrato nel capitolo precedente, LangChain mette a disposizione dello sviluppatore tutte le API necessarie per la realizzazione di chatbot che utilizzano LLM, semplificando lo sviluppo e garantendo un alto livello di astrazione.
A tal proposito, il caso di studio che l'azienda ha deciso di trattare, come anticipato in parte nei capitoli precedenti, è quello relativo alla progettazione e all'implementazione di un chatbot intelligente e interattivo, in grado di acquisire informazioni da un dataset di files PDF fornito in input, e rispondere, sulla base della conoscenza conseguita, ad eventuali query fornite dall'utente.
Le richieste principali erano fondamentalmente due: in primo luogo l'obiettivo principale era quello di riuscire ad implementare un buon RAG che consentisse al chatbot di rispondere in maniera coerente e puntuale in relazione al contenuto dei files presenti nel dataset e alla query fornita in input, limitando le possibili allucinazioni del modello; in secondo luogo, si voleva che il tutto venisse presentato mediante un'interfaccia grafica semplice e intuitiva, così da poter essere utilizzata in diversi ambiti e senza particolari conoscenze pregresse.
Per quanto riguarda lo sviluppo del RAG, e di conseguenza dell'intera pipeline, l'idea era quella di utilizzare le API Python di LangChain, soprattutto per una questione di comodità e praticità. Inoltre, da un punto di vista più tecnico e ai fini della tesi, un'ulteriore richiesta da parte dell'azienda era quella di includere e testare diverse tipologie di vectorDB, in modo da verificarne le performance e giungere ad alcuni

risultati in merito a ciò.
Durante la fase di realizzazione del chatbot, le esigenze aziendali hanno subito dei piccoli cambiamenti, in occasione di eventi e presentazioni ai clienti, il che ha portato alla creazione di più demo, che però verranno mostrate dettagliatamente nell'apposita sezione.

4.2 Requisiti

Nella fase preliminare allo sviluppo del software, oltre a condurre uno studio approfondito, volto ad acquisire nozioni relative a LLM, RAG e LangChain, è stato necessario determinare il materiale, gli strumenti e le tecnologie da utilizzare.
In uno stadio iniziale si è deciso di utilizzare un dataset temporaneo, costituito da files di tipo bancario, che poi è stato sostituito da un dataset reale, contenente dati assicurativi qualitativamente buoni.
Per quanto concerne gli LLM, dopo una fase iniziale in cui sono stati eseguiti dei test mediante modelli open-source di Hugging Face, l'azienda ha optato per l'utilizzo di modelli di OpenAI, sia per la creazione degli embeddings che per la generazione della risposta. Nello specifico, come anticipato nel capitolo precedente, sono stati utilizzati i modelli *text-embedding-ada-002* e *gpt-3.5-turbo*.
Inoltre, relativamente ai vectorDB, si è deciso di testarne alcuni open-source, come Chroma, FAISS e Databricks e uno proprietario, ovvero Pinecone.
Per la realizzazione delle demo e della user interface è stato impiegato Streamlit, un framework open-source che viene spesso impiegato nella realizzazione di applicazioni web scritte in Python. Esso è progettato per semplificare il processo di creazione di tali applicazioni, fornendo allo sviluppatore delle librerie standard e dei costrutti facilmente utilizzabili e ben documentati [14].
Infine, con l'intento di garantire una maggiore portabilità del codice, evitare problematiche legate a dipendenze presenti tra i molteplici pacchetti Python utilizzati ed avere un ambiente di esecuzione a sè stante, si è deciso di creare un enviroment Poetry. Esso non è altro che un tool open-source che agevola la gestione delle dipendenze e la creazione di virtual enviroments in Python ed è largamente utilizzato in ambito aziendale in quanto consente di semplificare il processo di gestione delle varie

librerie richieste da un determinato progetto, garantendo la riproducibilità dell'ambiente di sviluppo [15].

4.3 Demo

In merito alle demo del chatbot sviluppate, una prima versione è stata la seguente:

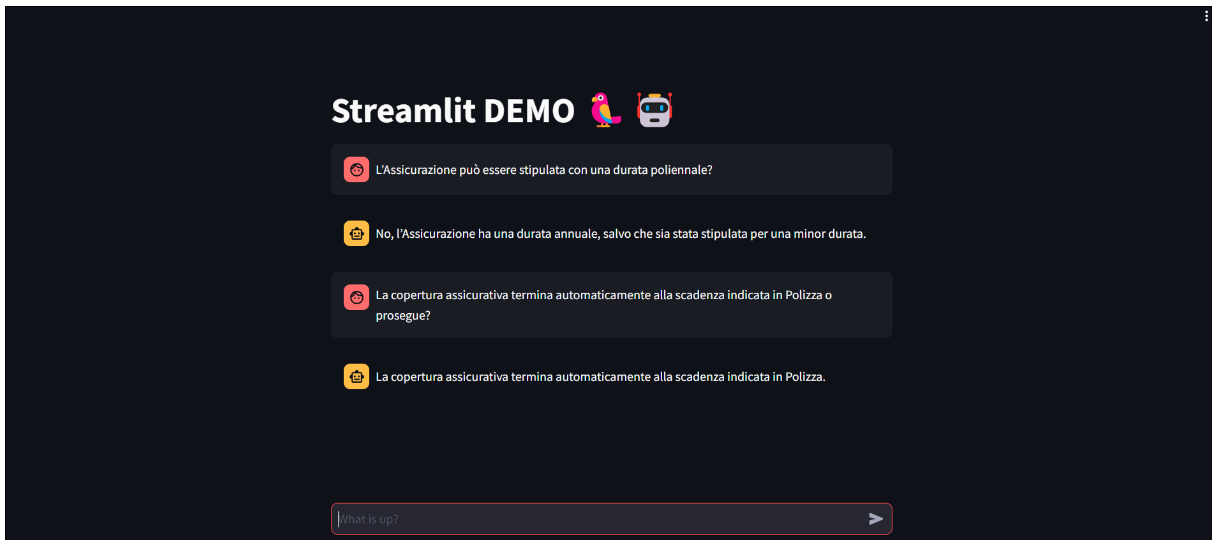

Figura 4.1: Versione iniziale del chatbot

Come si può notare, si tratta di un'interfaccia grafica molto semplice, ma funzionale. Sono stati utilizzati i costrutti messi a disposizione da Streamlit *chat_message* e *chat_input*. Il primo è stato utilizzato per visualizzare a video le domande poste dall'utente e le rispettive risposte fornite dal chatbot, mentre il secondo è stato impiegato per consentire l'inserimento della query. Per la generazione della risposta, è stato necessario incorporare la logica della pipeline descritta nella sezione 2.1 al suo interno. Nello specifico, è stato necessario passare attraverso tutti gli stadi, al fine di generare la risposta alla domanda dell'utente, che è stata poi aggiunta mediante un'*append* ai messaggi già presenti nel *session_state*. Proprio quest'ultimo è un modo che può essere utilizzato in applicazioni Streamlit per gestire informazioni e variabili relative alle singole sessioni utente.
In una fase successiva, è nata l'esigenza da parte dell'azienda di rivedere la demo implementata, in modo da arricchirla con altre funzionalità, in occasione della presentazione di quest'ultima ad un'azienda cliente. Fondamentalmente, le richieste erano: arricchire il chatbot con degli slider che consentissero di configurare alcuni

parametri in modo dinamico; fare in modo che i PDFs non fossero cablati all'interno del codice, bensì che fosse possibile inserirli al momento, tramite l'interfaccia grafica; infine, far sì che, nel momento in cui il chatbot non fosse in grado di reperire la risposta a partire dai file forniti, esso rispondesse in modo coerente, evitando allucinazioni da parte del modello.

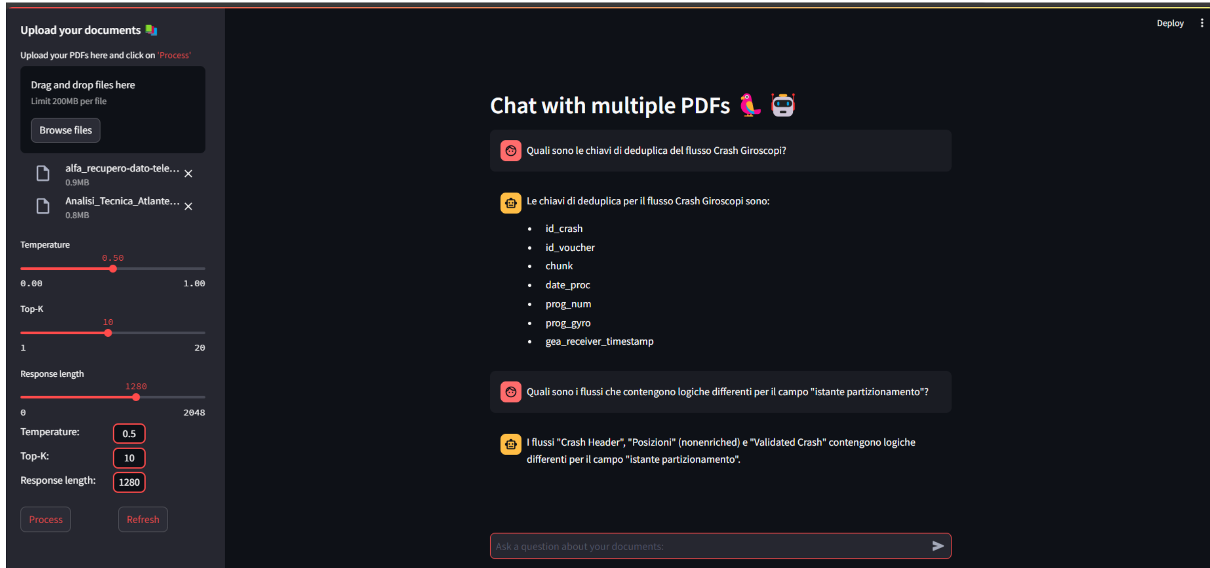

Figura 4.2: Seconda versione del chatbot con sidebar e sliders integrati

Dall'immagine soprastante si può osservare come sia stata aggiunta sulla sinistra una sidebar contenente diverse funzionalità. Innanzitutto, è stato utilizzato il costrutto *file_uploader* per far sì che l'utente riuscisse a caricare i propri PDFs. Inoltre, man mano che vengono aggiunti dei file, essi vengono visualizzati sotto la box di caricamento, in modo da gestirli ed eventualmente rimuoverli. In secondo luogo, sono stati inseriti tre slider che consentono di impostare dei parametri in tempo reale. A tal proposito, sono stati scelti la *Temperature*, la *Top-k* e la *Response length*. La *Temperature* è un parametro che viene utilizzato dagli LLM e che influisce sul processo di formulazione della response alla query da parte del modello. Entrando più nel dettaglio, in fase di generazione della risposta, ogni token, che viene utilizzato dal modello per "costruire" la risposta, è caratterizzato da una probabilità. Impostando una temperature bassa, si fa in modo che il modello vada a scegliere sempre il token successivo più probabile, il che porta ad avere risultati più deterministici. Viceversa, più alta sarà la temperature, più i risultati saranno "casuali", ovvero influenzati da una maggiore diversità e creatività [16]. La *Top-k* è un parametro che indica il numero di documenti più rilevanti che verranno

restituiti dal retriever, di cui il modello si serve ai fini della risposta. Infine, come si può intuire, la *Response length* va a definire il limite massimo in termini di lunghezza della risposta. I range scelti per i parametri citati sono 0.00-1.00 per la *Temperature*, 1-20 per la *Top-k* e 0-2048 per la *Response length*. Infine, nella sidebar sono state aggiunte delle text box che contengono i valori scelti dagli slider e due bottoni, uno per avviare il processing dei file caricati e uno per effettuare il refresh della chat history, utile nel caso in cui il chatbot intraprenda delle "strade sbagliate". Per quanto riguarda la parte relativa ai messaggi, essa è rimasta praticamente invariata rispetto alla versione iniziale, ad eccezione per il fatto che sono stati aggiunti alcuni controlli sulla *chat_input*, per fare in modo che l'utente fosse guidato all'utilizzo corretto del tool ed evitasse comportamenti errati, come ad esempio la formulazione di una query prima di aver inserito e avviato il processing di uno o più files.

In occasione di un evento, i cui partecipanti provenivano per lo più da ambiti bancari e assicurativi, l'azienda ha richiesto la realizzazione di un'ulteriore versione della demo, incentrata proprio su tali temi. Nello specifico, vi era l'esigenza di avere una pagina iniziale di carattere introduttivo, in cui venisse riportato un breve riassunto relativo al contenuto dei file, sia bancari che assicurativi, a partire dalla quale fosse poi possibile per l'utente accedere alla sezione desiderata, mediante due pulsanti.

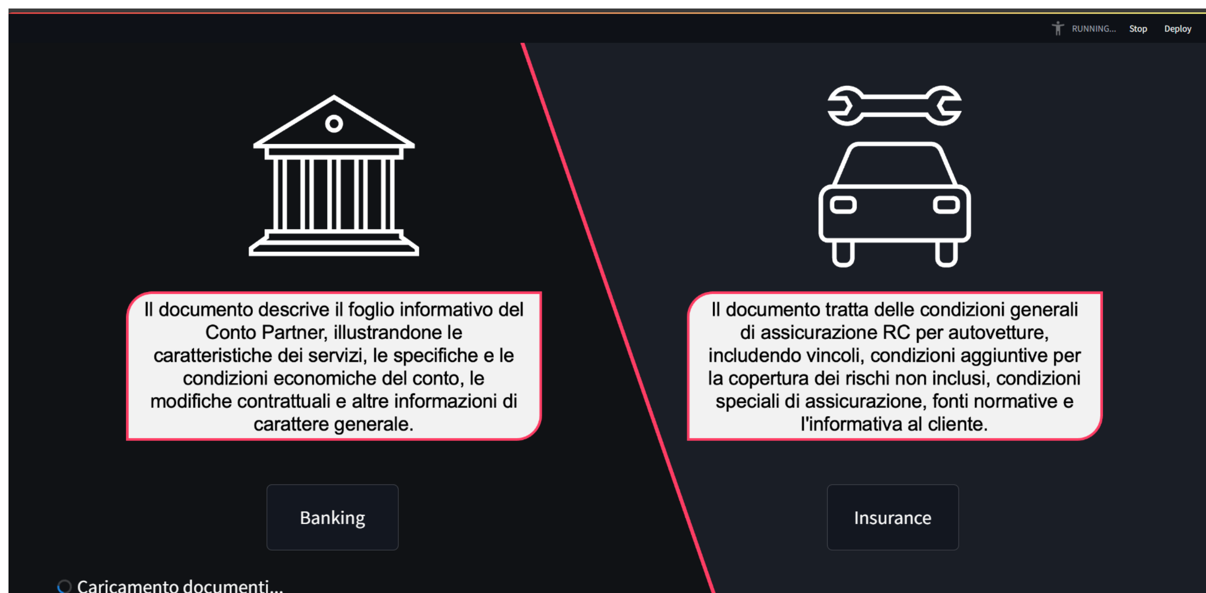

Figura 4.3: Home page della demo realizzata per l'evento

Di seguito è riportata la sezione "banking", che mette a disposizione un assistente virtuale bancario in grado di rispondere a domande inerenti agli argomenti trattati nei rispettivi

file. È stato integrato un costrutto *popover* di Streamlit per fare in modo che l'utente, cliccando, riuscisse ad accedere ad alcuni esempi di query da poter fornire al chatbot, in modo da rendere il tutto il più intuitivo possibile. La sidebar, che nella versione precedente conteneva gli slider e la box per l'upload dei PDFs, è stata svuotata e dotata unicamente delle pagine attraverso le quali è possibile navigare, oltre che a un pulsante per il refresh della *chat_history*. È stato inoltre fatto in modo che la risposta del chatbot a una determinata domanda contenesse anche il riferimento al file e alla pagina in cui sono contenute le informazioni reperite dal modello, per una questione di completezza. Per fare ciò, sono stati sfruttati i metadati contenuti nella lista di oggetti *Document* mediante i quali l'LLM formula la risposta.

Figura 4.4: Pagina dedicata all'ambito banking

È riportata, infine, la sezione "insurance", che presenta fondamentalmente le stesse funzionalità della sezione mostrata in precedenza, ad eccezione per il fatto che, per una questione di testing, si è deciso di integrare la risposta con un solo riferimento al file e alla pagina in cui le informazioni sono contenute, e non più di uno, come nel caso della sezione "banking".

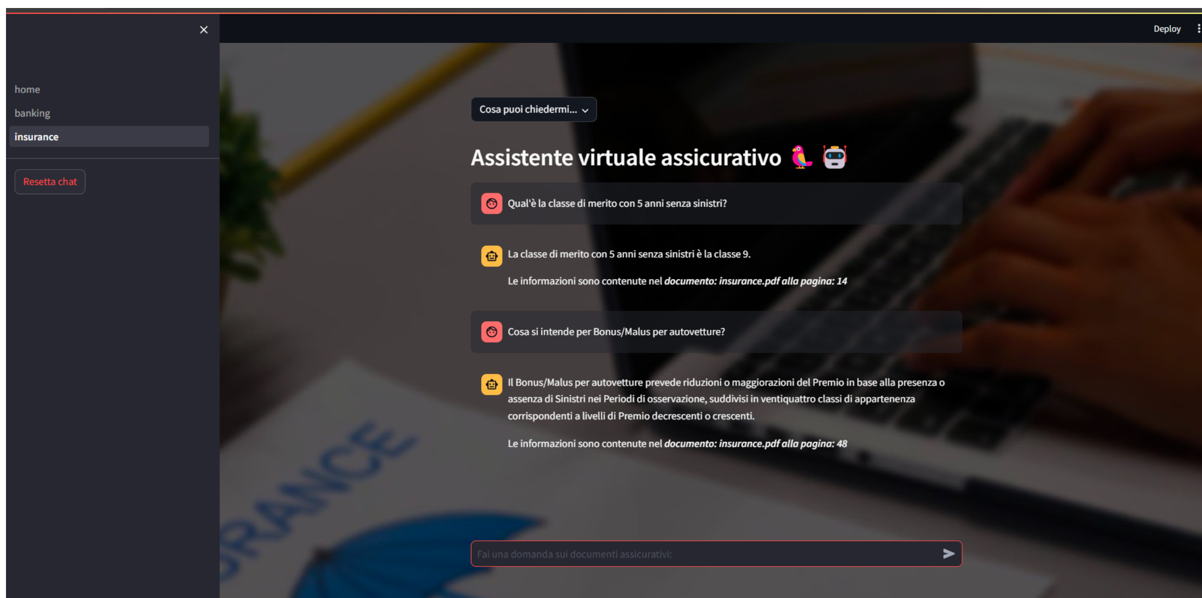

Figura 4.5: Pagina dedicata all'ambito insurance

Capitolo 5

Risultati sperimentali

5.1 VectorDB e tecniche di search

Per quanto riguarda l'evaluation e la raccolta di risultati in merito alle performance delle tecniche di search implementate per i vari vectorDB impiegati, si è deciso di utilizzare Ragas. Ragas (RAG Assessment) è un framework che consente di effettuare l'evaluation di una pipeline RAG e mette a disposizione una serie di tools mediante i quali è possibile raccogliere degli insights in merito alle performance di un RAG [27].

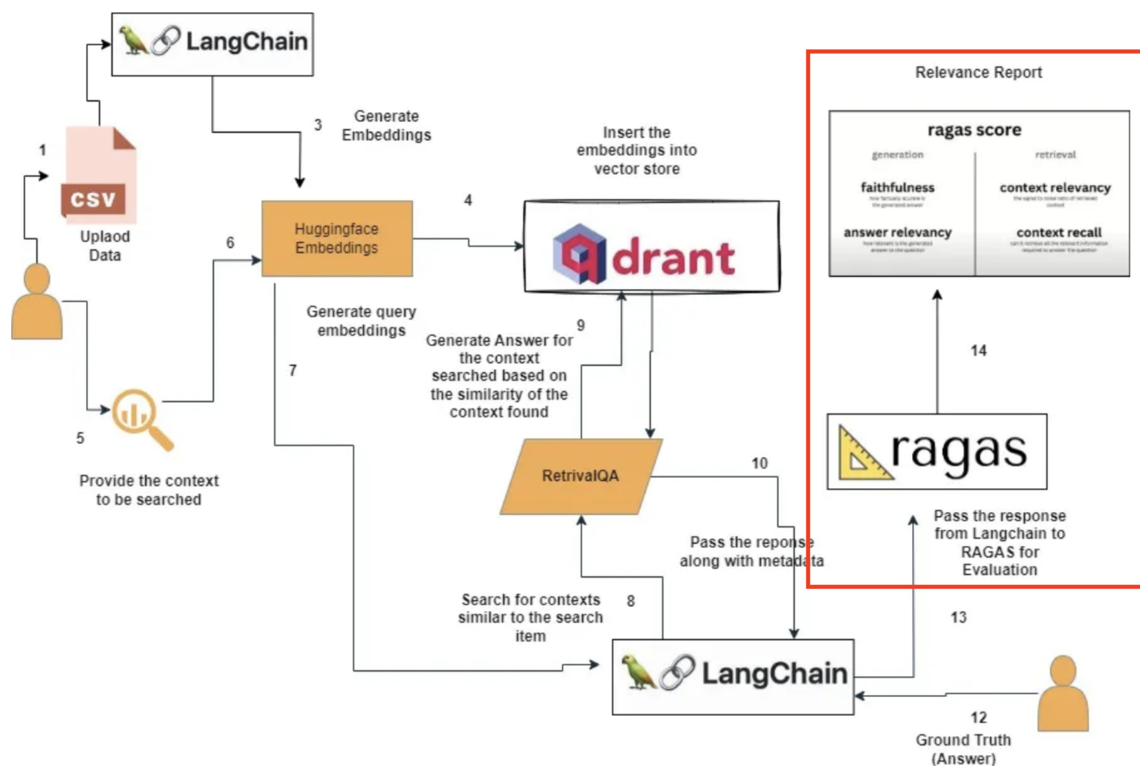

Figura 5.1: Esempio di pipeline in cui viene integrata l'evaluation con Ragas [28]

Prendendo spunto da alcuni articoli scientifici [29], si è optato per l'utilizzo del metodo *evaluation* di Ragas. Esso ha consentito di calcolare gli score a partire da un dataframe contenente le domande, le relative risposte, il contesto e la ground truth e da una lista di metriche su cui effettuare l'evaluation. È da premettere che tutte le informazioni relative a question, answer e ground truth sono state fornite in un apposito foglio di calcolo da parte dell'azienda cliente, in modo da poterle utilizzare per fini di testing e verificare il corretto funzionamento del RAG. Le metriche di Ragas che si è deciso di utilizzare sono: [26]

- **Faithfulness**: misura la coerenza della risposta generata rispetto al contesto fornito, può assumere valori nell'intervallo $(0, 1)$ e maggiore è il valore, migliore è la coerenza; è calcolata utilizzando la risposta e il contesto recuperato e la risposta generata è considerata coerente se tutte le affermazioni generate possono essere dedotte dal contesto fornito; il suo valore viene calcolato come il rapporto tra il numero di affermazioni nella risposta generata che possono essere dedotte dal contesto e il numero totale di affermazioni presenti nella risposta generata.

- **Answer relevancy**: misura la pertinenza della risposta generata rispetto al prompt fornito; valori più bassi indicano risposte incomplete o che contengono informazioni ridondanti, mentre valori più alti indicano una maggiore pertinenza; è calcolata a partire dalla query, dal contesto e dalla risposta generata, facendo una media delle similarità coseno medie della query originale rispetto a un numero N di domande artificiali, che a default è pari a 3.

- **Context recall**: misura fino a che punto il contesto recuperato si allinea con la risposta generata, che viene considerata come ground truth, può assumere valori nell'intervallo $(0, 1)$ e maggiore è il valore, migliore è la performance; è calcolata a partire dalla ground truth e dal contesto recuperato, mediante il rapporto tra le frasi della ground truth che possono essere attribuite al contesto e il numero di frasi presenti nella ground truth.

- **Context precision**: valuta se tutti gli elementi rilevanti della ground truth presenti nel contesto sono classificati o meno in posizioni prioritarie, in quanto, idealmente, tutti i chunks rilevanti dovrebbero avere i rank più alti; può assumere valori nell'intervallo $(0, 1)$ e maggiore è il valore, migliore è la precisione; è calcolata a partire dalla query, dal contesto e dalla ground truth attraverso il

rapporto tra la sommatoria delle precisioni dei vari chunks e il numero di elementi rilevanti nelle migliori K posizioni.

- **Context relevancy**: misura la pertinenza del contesto recuperato, calcolata sia sulla base della domanda che dei contesti; può assumere valori nell'intervallo $(0, 1)$ e più alto è il valore, maggiore è la pertinenza; si basa sul fatto che, idealmente, il contesto recuperato dovrebbe contenere esclusivamente informazioni essenziali per rispondere alla query fornita; è calcolata attraverso il rapporto tra il numero di frasi all'interno del contesto recuperato che risultano rilevanti per rispondere alla query fornita e il numero totale di frasi presenti nel contesto recuperato.

- **Context entity recall**: misura la context recall sulla base del numero di entità presenti sia nella ground truth che nei contesti rispetto al numero di entità presenti unicamente nella ground truth.

Una prima valutazione che è stata fatta, è quella relativa a un confronto tra la Keyword Search e la Semantic Search per i vectorDB Chroma e FAISS, che sono stati utilizzati maggiormente in una fase iniziale, complice il fatto che essi siano open-source e facilmente utilizzabili. Nella tabella sottostante sono stati riportati gli score calcolati sul Chroma vectorDB per le metriche sopracitate, in riferimento alla question "*Le nuove regole di sospensione valgono solo per la garanzia RCA?*".

	faithfulness	answer relevancy	context recall	context precision	context relevancy	context entity recall
KeywordSearch	NaN	0.0	0.0	0.36	0.07	0.0
SemanticSearch	1.0	0.0	1.0	0.23	0.0	0.0

Figura 5.2: Score calcolati sulla Keyword e Semantic Search del Chroma vectorDB

Come si può notare, la Semantic Search presenta degli score più alti in corrispondenza della faithfulness e alla context recall, mentre si comporta in maniera più o meno simile alla Keyword Search relativamente alle altre metriche, ad eccezione della context precision e context relevancy, in cui presenta degli score leggermente più bassi.
La medesima valutazione è stata fatta per il FAISS vectorDB, in questo caso in riferimento alla question "*Migrazione UniSalute: Le liste delle polizze da migrare su UniSalute, dove le posso trovare?*".

	faithfulness	answer relevancy	context recall	context precision	context relevancy	context entity recall
KeywordSearch	NaN	0.8939	1.0	0.0	0.0166	0.0
SemanticSearch	1.0	0.8938	1.0	0.0	0.0142	0.0

Figura 5.3: Score calcolati sulla Keyword e Semantic Search del FAISS vectorDB

Dalla tabella riportata si può osservare come il comportamento delle due tecniche di search sia abbastanza simile al caso precedente effettuato sul Chroma vectorDB. Nello specifico si può notare come la Semantic Search presenti uno score più alto in corrispondenza della faithfulness, mentre si comporti in maniera analoga alla Keyword Search in relazione ad altre metriche, ad eccezione di alcuni score che presentano delle leggerissime variazioni riguardo l'answer relevancy e la context relevancy.
Gli score presenti nelle due tabelle riportate evidenziano dei risultati che in parte già ci si poteva aspettare, ovvero che la Semantic Search, sfruttando l'embedding della query e di conseguenza il contesto di quest'ultima, presenti degli score più alti in corrispondenza della faithfulness e della context recall, che misurano rispettivamente la coerenza della risposta generata rispetto al contesto e l'allineamento del contesto con la risposta generata. D'altro canto, è altrettanto comprensibile il fatto che, la Keyword Search, sfruttando una o più parole chiave, presenti score più alti in relazione a metriche che misurano la precisione e la pertinenza del contesto.
Di seguito è stata riportata una tabella derivante da una valutazione fatta in seguito all'integrazione del Pinecone vectorDB. In essa sono stati riportati gli score complessivi calcolati sui vectorDB Chroma, Databricks e Pinecone. Gli score, in questo caso, non sono stati calcolati con Ragas mediante le metriche citate in precedenza, bensì sono stati calcolati da un'apposita funzione che consente di ricavare tale valore sulla base della posizione dei documenti recuperati. Nello specifico, il calcolo degli score è avvenuto prendendo in considerazione tutte le query presenti nel file fornito dall'azienda cliente. Innanzitutto è stato calcolato un valore temporaneo per ognuna delle suddette query mediante una funzione *calc_score*, dopodichè ne è stata fatta una media per ottenere il valore dello score complessivo per ognuna delle tecniche di search e per ognuno dei vectorDB. La funzione *calc_score* non fa altro che ricevere come parametri il nome del file associato alla query e la lista di documenti ottenuti in seguito alla fase di retrieval,

normalizzare le stringhe da cui essi sono costituiti, in modo da rimuovere eventuali caratteri speciali e, infine, restituire il valore calcolato. Per quanto riguarda il calcolo, tale funzione tiene conto fondamentalmente di due casistiche: nel caso in cui il documento indicato come ground truth non sia presente tra i k documenti più rilevanti ottenuti in fase di retrieval, verrà restituito 0, altrimenti il valore sarà calcolato nel seguente modo:

$$score = float\left(1 - \frac{i}{N}\right)$$

dove N è la lunghezza della lista di documenti rilevanti e i è l'indice del documento, che indica la posizione in cui esso si trova all'interno di tale lista.

Dall'algoritmo sopra riportato si evince come il valore dello score sia tanto più alto quanto più il rapporto tra i ed N sia basso. La logica adottata denota dunque score più alti nel caso in cui il documento associato alla query occupi posizioni più prioritarie nella lista e viceversa. Ovviamente, la posizione del documento non è casuale, bensì è strettamente correlata alla lista dei documenti ricavata a partire dal retriever e, di conseguenza, alle perfomance di quest'ultimo. Inoltre, il valore dello score è fortemente influenzato anche dalla lunghezza della lista, infatti più essa è lunga, minore sarà il rapporto tra i ed N, e quindi maggiore sarà lo score. In conclusione, da un punto di vista semantico, è possibile constatare come gli score calcolati non siano altro che degli indicatori di performance delle tecniche di search implementate per ciascun vectorDB.

Nella tabella non è stato riportato il FAISS vectorDB, in quanto, in seguito ad una fase iniziale in cui è emerso un comportamento molto simile a quello di Chroma, congiuntamente ad alcune indicazioni fornite dall'azienda, si è deciso di considerare Chroma come vector database open-source di riferimento, in base al quale effettuare dei confronti sugli ulteriori vectorDB aggiunti in fasi successive.

	Chroma	**Databricks**	**Pinecone**
KeywordSearch	0.172	0.032	0.468
HybridSearch	0.301	0.047	0.469
SemanticSearch	0.301	0.047	0.470

Figura 5.4: Score calcolati sulle tecniche di search implementate per Chroma, Databricks e Pinecone vectorDB

Come si evince dalla tabella, il vectorDB che presenta gli score più alti è Pinecone, seguito da Chroma e Databricks. Tali score mettono in evidenza dei risultati che ci si poteva aspettare, ovvero che Pinecone, grazie alla sua struttura interna che supporta sia vettori sparsi che densi e alla logica di gestione del vectorDB, offra prestazioni migliori rispetto a Chroma e Databricks.

5.2 Chunks

Un ulteriore test effettuato è relativo ai chunks. Nello specifico si è deciso di testare il comportamento delle tecniche di search in riferimento al Pinecone vectorDB, utilizzando diverse *chunk_size* e *chunk_overlap*.

Per quanto riguarda la dimensione dei chunk, non esiste una dimensione fissa che sia ottimale per qualsiasi use case, anzi, molto spesso, il modo migliore per determinare la dimensione migliore è mediante il testing. Attraverso degli esperimenti, è stato possibile effettuare un'evaluation delle performance con diverse *chunk_size* e trovare il miglior trade-off tra granularità e contesto [30]. Un ulteriore articolo reperito, a tal proposito, mette in evidenza come dimensione ed overlap dei vari chunks siano caratteristiche per le quali non è possibile fornire delle linee guida generali o dei valori specifici ottimali per ogni scenario, bensì essi sono strettamente correlati alla natura dei documenti, la tipologia di testo ed altri fattori, per via dei quali è spesso consigliabile procedere seguendo un approccio empirico [31].

È riportata ora la tabella ottenuta testando diverse *chunk_size*:

	100	300	512	768	1024
KeywordSearch	0.289	0.398	0.415	0.399	0.402
HybridSearch	0.305	0.402	0.417	0.406	0.410
SemanticSearch	0.415	0.412	0.424	0.424	0.435

Figura 5.5: Score calcolati utilizzando diverse chunk_size

Per le *chunk_size* testate, è stato preso spunto da alcuni articoli che suggerivano l'utilizzo di dimensioni multiple di 256 oppure, nel caso di chunk più piccoli, multiple di

100. Per questo, si è scelto di testare le *chunk_size* 100, 300, 512, 768 e 1024. Come si può osservare, la Keyword e la Hybrid Search presentano score più alti in corrispondenza di dimensioni medie, come 512. La Semantic Search, a differenza delle altre due tecniche, presenta delle performance migliori con l'utilizzo di chunk più grandi, come ad esempio di dimensione 1024.
Un test analogo è stato fatto sui *chunk_overlap*, prendendo come riferimento delle sovrapposizioni del 10%, 15% e 20%.

	10%	15%	20%
KeywordSearch	0.401	0.401	0.406
HybridSearch	0.408	0.407	0.411
SemanticSearch	0.434	0.434	0.438

Figura 5.6: Score calcolati utilizzando diverse chunk_overlap

Come ci si poteva aspettare, gli score migliorano all'aumentare della sovrapposizione, seppur in modo abbastanza modesto. A tal proposito, infatti, con un overlap del 20%, gli score aumentano per tutte le tre tecniche di search analizzate, seppur in maniera minima.

5.3 Soluzioni adottate per i pain points

D'ora in poi, saranno approfonditi i risultati inerenti all'applicazione delle diverse tecniche utilizzate per risolvere i pain points trattati nella sezione 3.3. L'analisi partirà dal cleaning dei dati mediante la libreria *unstructured.io*, che ha consentito di ottenere i seguenti score:

	Chroma	Chroma unstructured.io
KeywordSearch	0.172	0.173
HybridSearch	0.301	0.351
SemanticSearch	0.301	0.453

Figura 5.7: Score calcolati in seguito all'applicazione del cleaning con unstructured.io

Nella tabella sono stati messi a confronto gli score calcolati sul Chroma vectorDB originale e quelli calcolati sullo stesso database vettoriale, creato però in seguito alla pulizia dei dati messa in atto attraverso la suddetta libreria. Gli score ottenuti mettono in evidenza il miglioramento significativo delle performance della Semantic Search e score leggermente più alti rispetto all'originale per quanto riguarda la Hybrid Search. Tale comportamento può essere giustificato dal fatto che, grazie alla pulizia, le informazioni rilevanti presenti nel contesto consentono alla Semantic Search di migliorare l'accuratezza dei documenti restituiti, assegnando loro dei rank più prioritari rispetto a quelli assegnati dalla medesima search senza l'impiego del *cleaning con unstructured.io*. Allo stesso modo, l'Hybrid Search, essendo un ibrido delle altre due tecniche, beneficia, anche se in modo meno significativo, di tale operazione di pulizia. Infine, gli score della Keyword Search, come si poteva intuire, sono rimasti pressochè invariati. Da questa analisi, è possibile affermare che la tecnica di search che, comprensibilmente, ha beneficiato maggiormente del cleaning dei dati, è la Semantic.

Per quanto riguarda la valutazione in merito all'applicazione delle tecniche di Contextual Compression, Long Context Reorder e Reranking, si è optato di procedere nel seguente modo: si è deciso di prendere in considerazione questions, answer, context e ground truth relative alle query presenti nel file fornito dall'azienda cliente e calcolare gli score per le metriche descritte in precedenza in seguito all'applicazione delle tre tecniche citate; dopo aver ottenuto tali score, sono stati messi a confronto con quelli originali, ottenuti senza l'impiego di tecniche aggiuntive; in questo modo, per ogni score ottenuto in seguito all'impiego di ognuna delle tre tecniche tra Contextual Compression, Long Context Reorder e Reranking, è stato possibile verificare se ci fosse stato un miglioramento oppure un peggioramento.

Dal punto di vista pratico, è stata innanzitutto creata la tabella con gli score iniziali, quella con gli score ottenuti in seguito alla compressione del contesto, quelli ottenuti in seguito al "reordering" del contesto e, infine, quelli ottenuti in seguito alla riclassificazione dei documenti. Dopodichè, sono state confrontate singolarmente le tabelle relative alle tecniche con quella originale e ne sono stati esaminati i miglioramenti e i peggioramenti, per ogni specifico valore. È doveroso precisare che gli score sono stati calcolati sulla base delle tecniche di Ragas descritte a inizio capitolo.

Di seguito è riportato un grafico che raffigura l'andamento dei miglioramenti e dei

peggioramenti percentuali ottenuti in seguito alla Contextual Compression, in relazione al numero di query utilizzate:

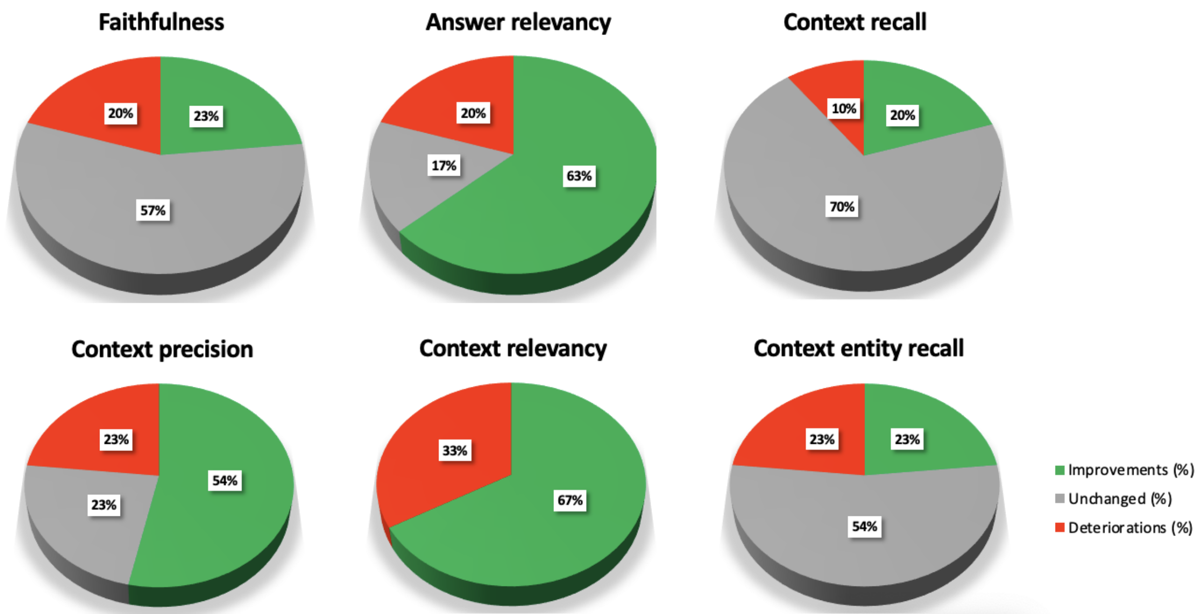

Figura 5.8: Analisi sull'impiego della Contextual Compression

Esaminando la figura, si può notare come l'impiego di questa tecnica abbia un impatto molto positivo, con miglioramenti degli score notevoli soprattutto in corrispondenza di metriche come answer relevancy e context relevancy. Proprio la context relevancy, infatti, aumenta quasi nel 70% delle query analizzate, in seguito alla Contextual Compression. I peggioramenti sono pochi in relazione ai miglioramenti e per lo più relativi alla context relevancy.
Verrà riportato ora l'analogo grafico, ottenuto in seguito all'applicazione del LongContextReoder.

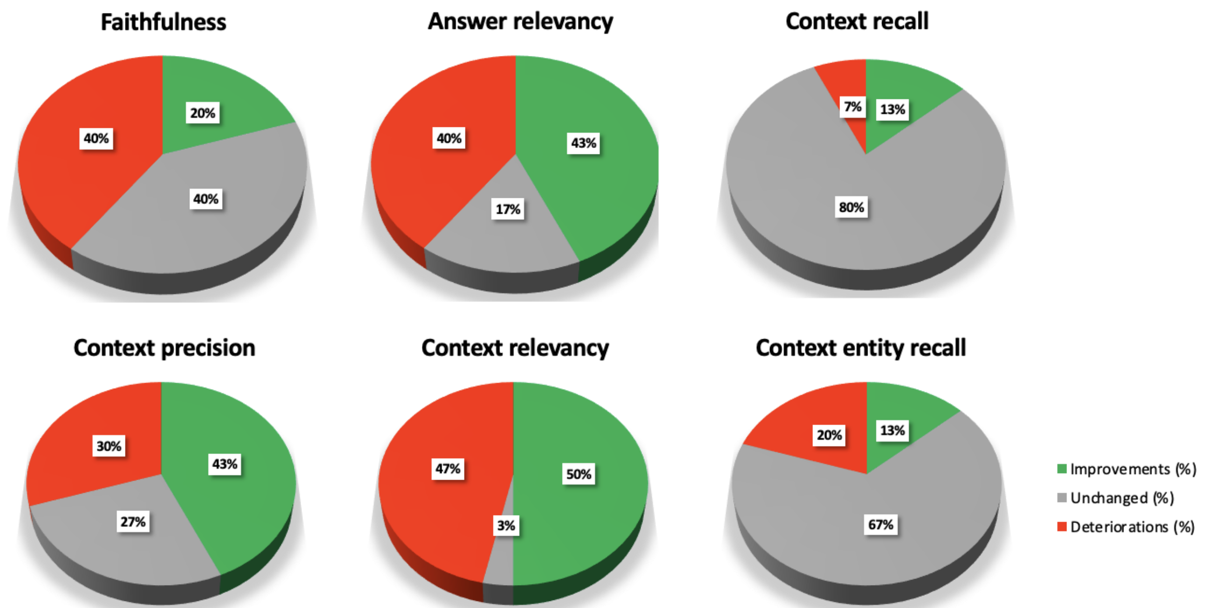

Figura 5.9: Analisi sull'impiego del Long Context Reorder

Al contrario del caso precedente, si osserva che i peggioramenti e i miglioramenti hanno un andamento molto simile e quasi si equivalgono. Ciò denota che l'impatto del Long Context Reorder non apporta benefici significativi al RAG e che le prestazioni rimangono pressochè invariate.

Per quanto riguarda il Reranking, sono stati studiati i medesimi trend in relazione all'impiego del FlashrankRerank e del Ranker, separatamente.

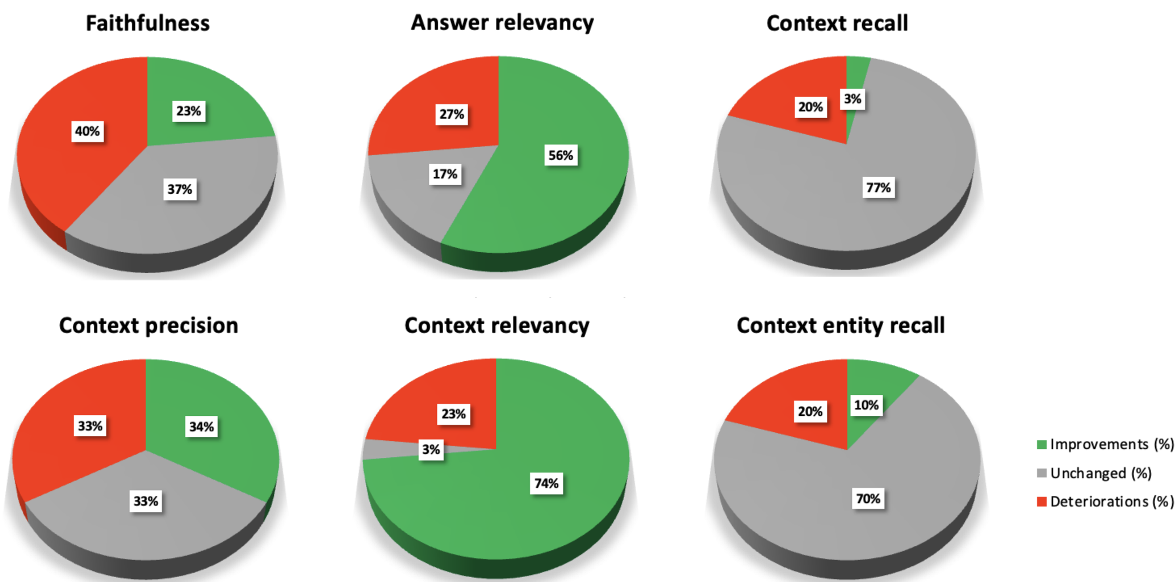

Figura 5.10: Analisi sull'impiego del Reranking con FlashrankRerank

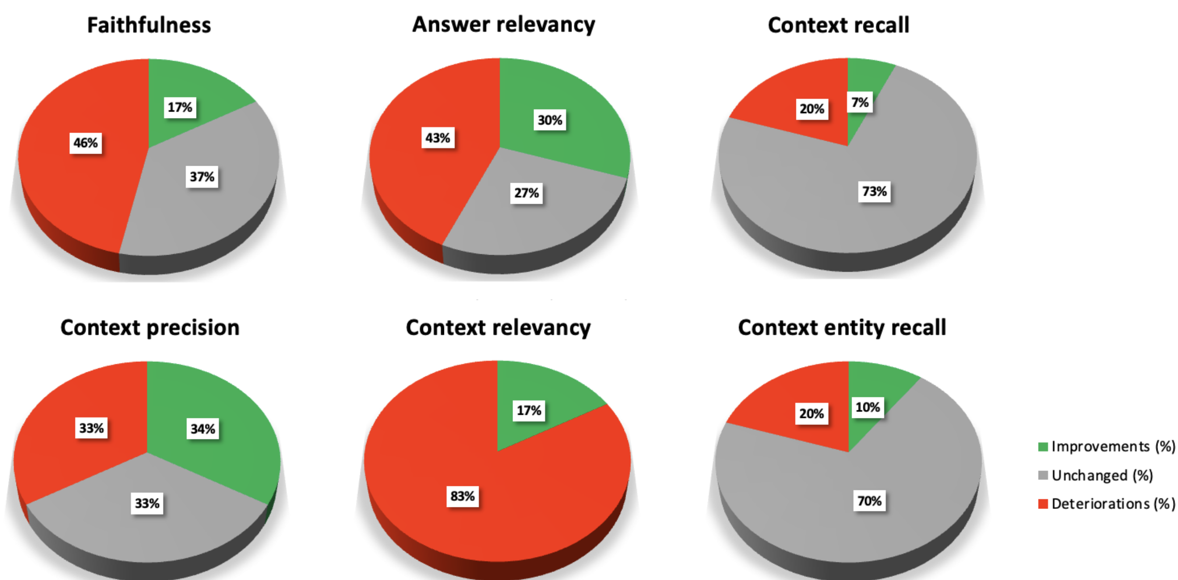

Figura 5.11: Analisi sull'impiego del Reranking con Ranker

Dai due grafici riportati, emerge come il FlashrankRerank apporti maggiori benefici rispetto al Ranker. Esso infatti consente di avere una percentuale di miglioramento che si avvicina al 60% nel caso dell'answer relevancy e superiore al 70% nel caso della

context relevancy, mantenendo le percentuali di peggioramento relativamente basse per tutte le metriche. Al contrario, il grafico inerente all'impiego del Ranker segnala che, in proporzione, la percentuale dei peggioramenti è molto più alta rispetto a quella dei miglioramenti. In particolare, si può notare come gli score della context relevancy siano peggiorati in più dell'80% dei casi in esame.

Complessivamente, dallo studio dei grafici riportati, è possibile affermare che le tecniche che apportano maggiori benefici al RAG dal punto di vista dell'ottimizzazione delle performance sono la Contextual Compression e il Reranking. Nello specifico, per quanto concerne il Reranking, l'utilizzo del FlashrankRerank offre prestazioni nettamente superiori rispetto all'impiego del Ranker.

Conclusioni

Questo elaborato di tesi mira ad analizzare e trattare argomenti quali Generative AI, LLM e LangChain e i punti essenziali per la realizzazione di un RAG, che consenta di sviluppare un prototipo di chatbot in grado di rispondere alle query di un utente, sulla base delle informazioni contenute all'interno di un dataset di files PDF.

In primis, è stata fornita una panoramica generale sugli argomenti precedentemente citati, sul loro funzionamento, i loro impieghi e la loro importanza crescente nel panorama mondiale. In secondo luogo, è stato introdotto il concetto di RAG come metodologia che consente il recupero delle informazioni e la generazione della risposta, descrivendo inoltre gli stadi della pipeline mediante la quale esso viene implementato. Successivamente, è stata trattata in modo approfondito l'implementazione software del progetto trattato in questa tesi, analizzando nel dettaglio le API e le componenti utilizzate. Sono state esplorate in modo esaustivo le tecniche di search implementate per i vectorDB impiegati, così come le soluzioni che si propongono di risolvere alcuni pain points comuni ai RAG. Sono stati descritti il caso di studio e i requisiti alla base del progetto e sono state mostrate le diverse versioni di demo implementate. Infine, sono stati riportati i risultati derivanti da alcuni test effettuati in merito ai vectorDB e alle tecniche di search, ai chunks e alle soluzioni adottate per i pain points.

In special modo, a conclusione dell'elaborato, è bene soffermarsi proprio su tali test. Come riportato nel capitolo precedente, da essi emerge come Pinecone sia il database vettoriale, tra quelli impiegati in fase di sviluppo, che offre le prestazioni migliori. Ciò deriva senza dubbio dal fatto che esso possiede una struttura di supporto che consente una gestione migliore dei dati memorizzati e, di conseguenza, di ottenere un RAG con performance migliori. Per quanto riguarda i chunks, è doveroso sottolineare come non esistano delle linee guida univoche che determinino la perfetta *chunk_size* e il perfetto *chunk_overlap*, bensì esse dipendano dalla tipologia dei documenti in questione, dagli

emebddings, dal vectorDB in cui verranno caricati e da una serie di altri fattori. Per questo motivo, è necessario adottare un approccio empirico, al fine di determinare la dimensione e la sovrapposizione ideali per lo use case in questione.
In chiusura, dai test effettuati è emerso come la Contextual Compression e il Reranking siano delle tecniche che, nel progetto in questione, hanno apportato i maggiori benefici in termini di performance del RAG, al contrario del Long Context Reorder, che non ha apportato i benefici sperati. Per quanto concerne il Reranking, è stato possibile evidenziare come l'utilizzo del FlashrankRerank abbia apportato una percentuale di miglioramenti più significativa rispetto all'impiego del Ranker.

Sviluppi futuri

Il progetto trattato in questa tesi si adatta bene ad eventuali sviluppi futuri, in quanto potrebbe essere esteso in diverse direzioni.
Lato embeddings e vectorDB si potrebbero impiegare e testare altri modelli di embeddings, come ad esempio BERT o Word2Vec, e altri vectorDB, come quello messo a disposizione da Azure o MongoDB, ampiamenti utilizzati in ambito aziendale.
Lato prompting invece, si potrebbero studiare delle tecniche di prompting che contribuiscano ad aumentare l'accuratezza, si potrebbe valutare l'impatto della qualità dell'input sulle risposte generate e si potrebbe lavorare in merito alla qualità e alla tipologia delle risposte generate dal modello. A tal proposito, uno degli obiettivi futuri dell'azienda, è proprio quello di estendere il progetto discusso in termini di lessico della risposta generata, in modo che il chatbot possa adattarsi a qualunque contesto, impiegando termini più o meno tecnici a seconda della natura dei documenti e dello use case considerato.

Ringraziamenti

Questo spazio dell'elaborato è dedicato alle persone che hanno contribuito, con il loro supporto, alla realizzazione dello stesso.
In primis, ringrazio il mio relatore Prof. Matteo Poggi, per la sua disponibilità, per i suoi consigli e per avermi dato la possibilità di trattare questo argomento.
Ringrazio la mia famiglia, i miei amici e tutti coloro che mi sono stati vicini in questi anni.
Desidero esprimere la mia profonda gratitudine ai miei genitori e a mio fratello, che mi hanno sostenuto fin dal primo giorno e hanno sempre creduto in me.
Un ringraziamento speciale va ad Alice, che ha condiviso con me la parte finale di questo percorso, offrendomi il suo supporto e incoraggiamento sempre, in qualsiasi momento.
Ultimo, ma non per importanza, a Manuel Greco, con il quale ho avuto la possibilità di collaborare per la riuscita di questo progetto.

Bibliografia

[1] What is Generative AI? *https://www.oracle.com/it/artificial-intelligence/generative-ai/what-is-generative-ai/*

[2] Economic potential of generative AI *https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/the-economic-potential-of-generative-ai-the-next-productivity-frontier*

[3] Generative Artificial Intelligence: Trend and Prospects *https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9903869*

[4] What are Large Language Models? *https://aws.amazon.com/it/what-is/large-language-model/*

[5] What is LangChain? *https://www.ibm.com/it-it/topics/langchain/*

[6] What is LangChain? *https://aws.amazon.com/it/what-is/langchain/*

[7] LangChain documentation *https://python.langchain.com/v0.2/docs/introduction/*

[8] Using LangChain for Question Answering on Own Data *https://medium.com/@onkarmishra/using-langchain-3af0a82789ed*

[9] Enhancing Semantic Search with LangChain, Vector Databases, and Llama2–70B-Chat *https://medium.com/@alroumi.abdulmajeed/*

[10] What is Retrieval-Augmented Generation? *https://aws.amazon.com/it/what-is/retrieval-augmented-generation/*

[11] Design Patterns in Python: Factory Method *https://medium.com/@amirm.lavasani/design-patterns-in-python-factory-method-1882d9a06cb4*

[12] Automatic Prompt Engineer (APE) *https://www.promptingguide.ai/it/techniques/ape*

[13] Large Language Models Are Human-Level Prompt Engineers *https://arxiv.org/abs/2211.01910*

[14] Streamlit documentation *https://docs.streamlit.io*

[15] Poetry documentation *https://python-poetry.org/docs/*

[16] LLM Settings *https://www.promptingguide.ai/it/introduction/settings*

[17] Hybrid Search for Retrieval Augmented Generation (RAG): a more effective search *https://medium.com/@lydiaarezkilydia/*

[18] Understanding hybrid search *https://docs.pinecone.io/guides/data/understanding-hybrid-search*

[19] 12 RAG Pain Points and Proposed Solutions *https://towardsdatascience.com/12-rag-pain-points-and-proposed-solutions-43709939a28c*

[20] Unstructured.io library *https://unstructured.io*

[21] Contextual Compression *https://blog.langchain.dev/improving-document-retrieval-with-contextual-compression/*

[22] Long Context Reorder *https://arxiv.org/abs/2307.03172*

[23] Reranking *https://medium.com/@ashpaklmulani/improve-retrieval-augmented-generation-rag-with-re-ranking-31799c670f8e*

[24] OpenAI Embeddings *https://platform.openai.com/docs/guides/embeddings/what-are-embeddings*

[25] Unleashing the Power of Prompt Engineering: Zero-Shot, One-Shot, and Few-Shot Inference *https://medium.com/@S.Shakir/*

[26] Ragas Metrics *https://docs.ragas.io/en/stable/concepts/metrics/index.html*

[27] Ragas Documentation *https://docs.ragas.io/en/stable/*

[28] Evaluate RAG Pipeline using RAGAS *https://medium.aiplanet.com/evaluate-rag-pipeline-using-ragas-fbdd8dd466c1*

[29] Evaluating RAG Applications with RAGAs *https://towardsdatascience.com/evaluating-rag-applications-with-ragas-81d67b0ee31a*

[30] How to determine the perfect chunk size? *https://medium.com/@farenas1/*

[31] Chunk Division and Overlap: Understanding the Process *https://gustavo-espindola.medium.com/chunk-division-and-overlap-understanding-the-process-ade7eae1b2bd*