UNIVERSITÀ DEGLI STUDI DI
NAPOLI FEDERICO II

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

Elaborato finale in Big Data Engineering

LLM2Query: Query del Cuore

Anno Accademico 2024/2025

Relatore
Prof. Vincenzo Moscato

Candidato
Vincenzo Luigi Bruno matr. M63001670
Salvatore Cangiano matr. M63001647

Abstract

La gestione di dati complessi e non strutturati in cardiologia impegna i medici in inefficienti estrazioni manuali. Il progetto LLM2Query presenta un prototipo per superare questa criticità, minimizzando i tempi di recupero delle informazioni. LLM2Query democratizza l'accesso ai database medici fornendo un'interfaccia in linguaggio naturale che astrae la complessità tecnica dell'interrogazione. Ciò consente ai cardiologi, anche senza competenze informatiche specifiche, di formulare query per database MongoDB. Questa traduzione è affidata a un motore basato sull'API Gemini di Google e potenziato da un approccio Retrieval Augmented Generation (RAG) per un'accurata interpretazione del lessico medico. Il prototipo è stato sviluppato e testato su un dataset reale di visite mediche anonimizzate della regione Campania. L'obiettivo è restituire ai medici tempo per l'analisi clinica e la cura del paziente, semplificando l'accesso ai dati.

Contents

Chapter 1

Preprocessing & Caricamento del Dataset

1.1 Composizione del Dataset

Il dataset di partenza raccoglie informazioni cliniche eterogenee di pazienti riguardanti visite effettuate in Campania. Per questo progetto, ci si è focalizzati su un sottoinsieme specifico di tabelle rilevanti per il dominio cardiologico.

Le due tabelle principali sono *ANAGRAFICA* e *LISTA_EVENTI*.

La tabella *ANAGRAFICA* contiene tutte le informazioni demografiche e sanitarie di base dei pazienti. Questa tabella è cruciale in quanto il campo *ID_PAZ* (derivato da *SEZIONE* e *CODPAZ*) viene utilizzato come chiave primaria per collegare i dati anagrafici alle altre tabelle del dataset.

La tabella *LISTA_EVENTI* funge invece da indice cronologico, registrando tutti gli eventi clinici occorsi ai pazienti nel tempo. È importante notare che questa tabella non include la totalità degli eventi presenti nel DB di partenza, ma si concentra su quelli relativi a tabelle specifiche come *ANAMNESI*, *CORONAROGRAFIA_PTCA*, *ECOCARDIO_DATI*, *ECOCAROTIDI*, *ESAMI_LABORATORIO*, *RICOVERO_OSPEDALIERO* e *VISITA_CONTROLLO_ECG*. Ogni evento in questa tabella è caratterizzato da un *ID_PAZ* (codice univoco), una *DATA* dell'evento e un *TIPO_EVENTO* che specifica la tabella di origine dei dati dettagliati.

Per ulteriori dettagli si rimanda alla visione della documentazione ufficiale del dataset.

1.2 Preprocessing e Caricamento dei Dati

Per garantire uniformità e facilitare le future interrogazioni è stata effettuata una fase di pulizia e trasformazione dei dati.

Si è proceduto con:

- **Normalizzazione del casing**: Per alcune colonne testuali (*COGNOME*, *NOMEPAZ*, *COMUNE_DI_NASCITA*) nella tabella *ANAGRAFICA*, i valori sono stati convertiti in maiuscolo per garantire uniformità e facilitare le future interrogazioni.

- **Gestione dei valori nulli**: Diversi valori eterogenei indicanti

assenza di dato (es. stringhe vuote, "-", "null", "NaN") sono stati normalizzati al valore None (nullo) in tutte le colonne di tutti i DataFrame caricati.

- **Anonimizzazione dei dati sensibili**: Per proteggere la privacy dei pazienti, è stata implementata un procedura di anonimizzazione. Questa funzione modifica le colonne contenenti dati personali:
  - I nomi ed i cognomi sono stati sostituiti cono la sola lettera iniziale.
  - Per il codice fiscale venogno sostituiti tutti i caratteri alfabetici con un trattino, offuscando quindi tutte le parti del codice legate a nome, cognome e data di nascita, ma preservandone la struttura.
- **Caricamento su MongoDB Atlas**: Una volta processati, i DataFrame Spark sono stati caricati come collection nel database di destinazione.

1.3 Embedding della Documentazione del Dataset

Per fornire al LLM il contesto necessario per interpretare correttamente le query in linguaggio naturale, è stata creata una documen-

tazione dettagliata per ciascuna tabella del dataset rilevante.

Questa fase, ha comportato:

1. **Preparazione dei file di documentazione**: per ogni tabella significativa è stato redatto un file di testo che descrive:
    - Lo **scopo** della tabella.
    - Le **relazioni chiave** con altre tabelle.
    - Un elenco dettagliato dei vari **campi**, con il nome, la descrizione del contenuto, il tipo di dato e valori di esempio.
    - **Logica di business** incorporata nella tabella.
    - **Esempi comuni di utilizzo nelle query** per quella specifica tabella.
2. **Generazione degli Embedding e archiviazione**: Il testo di ogni documento è stato convertito in embedding. Per ogni tabella, l'embedding generato dal suo file di documentazione è stato aggiunto a una collection di un'istanza di database vettoriale ChromaDB. Insieme all'embedding, sono stati memorizzati il documento testuale originale e metadati utili come il nome del file sorgente ed il nome della tabella a cui si riferisce.

Questo processo di embedding della documentazione è fondamentale per il funzionamento del RAG, dato che ogni query formulata dall'utente viene anch'essa trasformata in un embedding ed utilizzata

per interrogare ChromaDB. Il database restituisce dunque i documenti semanticamente simili alle query dell'utente e questi vengono poi forniti come contesto aggiuntivo all'LLM, che li utilizza per generare una query più precisa e pertinente.

Chapter 2

Soluzione Proposta: Query del Cuore

Il progetto **Query del Cuore** propone un approccio basato su un agente intelligente. L'idea centrale è permettere agli utenti di esprimere le proprie esigenze informative in **linguaggio naturale**, senza dover conoscere complesse sintassi di interrogazione. L'agente è progettato per interpretare queste richieste e tradurle automaticamente in query sul database. Prima dell'esecuzione, le query generate vengono sottoposte a un'analisi preventiva, garantendo accuratezza e pertinenza dei risultati.

Per immagazzinare i dati sensibili delle visite dei pazienti, il progetto Query del Cuore ha scelto **MongoDB**. Questo database NoSQL, noto per la sua flessibilità e scalabilità, si adatta perfettamente alla natura eterogenea dei dati clinici, consentendo una gestione efficiente

e dinamica delle informazioni.

Il modello LLM utilizzato che dà vita la nostro agente intelligente è il modello **gemini2.0flash**. Il modello non viene alterato o "finetunato" per il task in questione (generazioni di query); si fa utilizzo di un approccio **RAG** (Retrieval Augumented Generative) per migliorare le prestazioni del modello.

2.1 Architettura del Sistema

In questo paragrafo verrà descritta l'architettura del sistema, compresi i flussi operativi principali e le tecnologie associate.

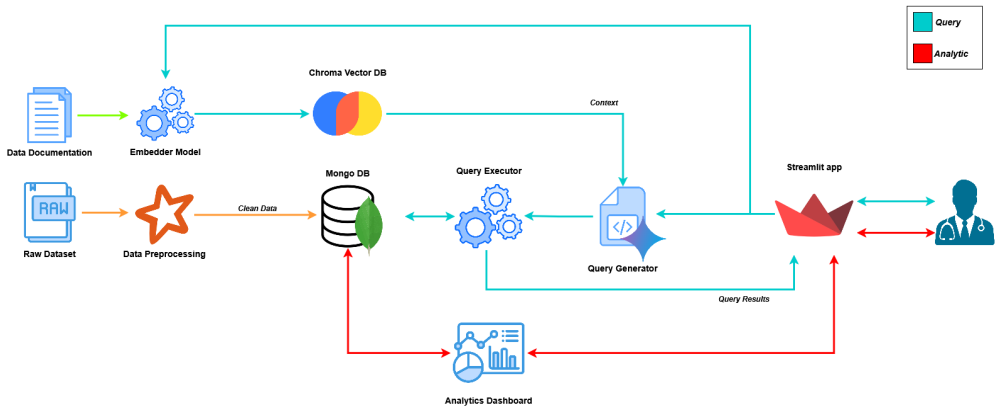

Figure 2.1: Architettura Query del Cuore

I flussi operativi verranno descritti dettagliatamente nei prossimi paragrafi, insieme però descrivono il "**datapath**" del sistema. Possiamo individuare quattro flussi nel nostro sistema:

- Flusso Preprocessing Dati
- Flusso creazione Embedding vectors
- Flusso Creazione ed Esecuzione Query
- Flusso Analitiche e UI utente

2.1.1 Flusso Arancione: Preprocessing dei Dati

Il processo di preparazione e gestione dei dati all'interno del nostro sistema inizia con l'acquisizione di un vasto corpus di informazioni mediche, originariamente strutturate in file CSV. Questi dataset sono stati inizialmente importati e gestiti in un ambiente di sviluppo **Google Colab**. Per garantire un'elaborazione efficiente e scalabile di volumi significativi di dati, abbiamo fatto ampio ricorso a **PySpark**, la libreria Python per Apache Spark. Questa scelta ha permesso di sfruttare le capacità di elaborazione distribuita di Spark, ottimizzando le performance in tutte le fasi del pipeline di dati.

Successivamente all'ingestione, i dati sono stati sottoposti a un'accurata serie di operazioni di pulizia e trasformazione, essenziali per assicurare l'integrità, la coerenza e l'utilizzabilità delle informazioni. Le fasi principali di questa pulizia sono state tre:

- **Creazione di ID univoco tra le collezioni**
- **Trimming e Uppercase delle Colonne Testuali**

- **Standardizzazione dei Valori Nulli**
- **Anonimizzazione del Dataset**

Conclusa questa fase di rigorosa pulizia e preparazione, i dati sono stati caricati nel nostro database. Sfruttando l'efficiente connettore messo a disposizione da PySpark per MongoDB, e opportunamente configurato per ottimizzare il trasferimento e l'indicizzazione, le informazioni elaborate sono state migrate con successo su un cluster MongoDB Atlas.

2.1.2 Flusso Verde: Creazione dei vettori di Embedding

Questo flusso operativo gioca un ruolo cruciale nella fase di preparazione del contesto informativo per il nostro agente intelligente. Il suo obiettivo principale è quello di trasformare la documentazione descrittiva del nostro database, inizialmente composta da file di testo ('.txt') che illustrano lo schema delle tabelle MongoDB, in un formato semanticamente comprensibile per il modello LLM.

Il processo si articola in diverse fasi:

- **Caricamento della Documentazione**
- **Generazione degli Embedding**
- **Costruzione dell'Indice Vettoriale**

- **Memorizzazione Persistente**

Abbiamo utilizzato il modello ***gte-large***, disponibile tramite la libreria sentence-transformers di **Hugging Face**, per generare gli embedding vettoriali dei nostri documenti descrittivi. Data la dimensione contenuta di ogni documento, un singolo vettore è sufficiente per catturare e rappresentare efficacemente il suo contenuto informativo completo. Il tutto viene infine salvato in modo persistente in una **chroma_collection**.

Trasformando descrizioni testuali in un formato vettoriale interrogabile, il flusso abilita un recupero di informazioni contestuali preciso ed efficiente, migliorando drasticamente la capacità del modello **gemini-2.0-flash** di generare query pertinenti e accurate in risposta alle esigenze espresse in linguaggio naturale dall'utente.

2.1.3 Flusso Azzurro: Creazione ed Esecuzione delle query

Questo flusso operativo è probabilmente il più complesso dell'intero sistema. Difatti possiamo dividere il processo di costruzione ed esecuzione query in tre fasi sequenziali:

1. Creazione del vettore di Embedding della query utente (in linguaggio naturale)

2. Estrazione dei documenti più rilevanti rispetto alla query utente.

3. Creazione del contesto per il generatore di query (RAG)

4. Generazione della query pyMongo

5. Preprocessing ed esecuzione della query generata

6. Restituzione dei risultati estratti da MongoDB all'utente

E' possibile iniziare questo flusso a partire dalla funzionalità principale del sistema: la modalità "**Assistente**". Ogni fase del processo è svolta da uno o più moduli del progetto che verranno analizzati nel dettaglio negli appositi paragrafi successivi.

2.1.4 Flusso Rosso: Analitiche descrittive del Dataset

L'ultimo flusso del sistema possiamo dire che è quello più "classico" che implementa delle funzionalità tipiche di una applicazione di infotainment.

Tramite altre viste del sistema come **Analitche** e **Cartella Clinica Paziente**, l'utente può inserire degli input con i quali il sistema potrà eseguire delle query prestabilite per creare grafici e tabelle che generano valore e informazione utili per il medico.

2.2 Moduli del Progetto

Il progetto LLM2Query è stato concepito con un' **architettura modulare**, un approccio che garantisce non solo chiarezza nella struttura

del codice, ma anche una netta separazione delle responsabilità tra i diversi componenti. Questa impostazione è fondamentale per facilitare la manutenzione del sistema e per assicurare una **scalabilità** efficiente in futuro.

Ci sono tre package python su cui porremo il nostro interesse:

- preprocessing
- query engine
- analytics
- evaluation

2.2.1 Package Preprocessing

In questo package sono presenti i due python notebook usati per implementare i due flussi operativi per la preparazioni dei dati. In particolare:

- **Data_Extraction.ipynb**: implementa una classe **CSVLoaderManager** che astrae un curatore del dataset clinico, implementa metodi per la pulizia e il caricamento dei dati su Mongo, sfruttando la libreria **pyspark.sql**.
- **EmbeddingDatasetDoc.ipynb**: il notebook racchiude il processo di trasformazione dei documenti che descrivono il dataset in vettori di embedding. Questi vettori insieme alla collection

chroma verranno utilizzati da altri moduli per estrarre i documenti ritenuti più utili per la creazione della query.

2.2.2 Package Query engine

Questo Package può essere considerato il motore fondamentale della funzione principale, ossia la generazione automatica della query pymongo.

- **query_generator.py**: all'interno di questo modulo vi è la classe **MongoDBQueryGenerator**, l'astrazione del generatore delle query, viene inizializzato con alcuni strumenti necessari:
  - Un modello di embedding (gtelarge), necessario alla conversione delle iscrizioni dell'utente in rappresentazione vettoriale
  - Un'istanza di LLM fornita da **google.generativeai** (Gemini)
  - Un client ChromaDB per interagire con il database vettoriale persistente, anche per fare la query di similarità tra il vettore query utente e i vettori dei documenti.
  - Lo schema del database MongoDB, utile per aumentare il contesto di conoscenza del nostro agente intelligente.

  La funzione **retrieve_context** esegue una ricerca di similiratà nel database vettoriale, i documenti estratti verranno utilizzati

per arricchire il **prompt di sistema**. Si può configurare anche il numero di documenti estratti che di default è settato a tre.

La funzione **generate_query** costruisce un prompt dettagliato che include l'istruzione dell'utente, lo schema del database e il contesto recuperato. Il prompt è studiato per guidare l'LLM a produrre un output JSON specifico, indicando chiaramente il formato atteso per query find o aggregate, e gestendo anche i casi di richieste irrilevanti.

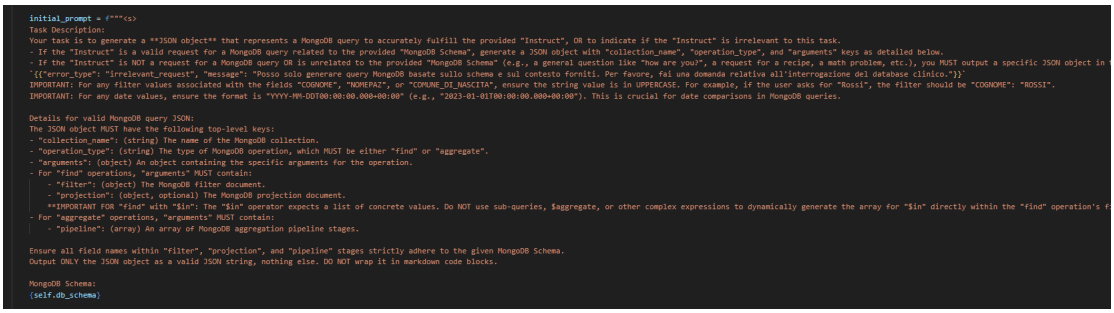

Figure 2.2: Prima parte del system Prompt

Il prompt è stato costruito anche immettendo degli esempi di documenti json corretti, adottando così un approccio di **few shots learning** (figura 2.3. Alleghiamo anche uno screen (in figura 2.4) di come vengono immesse le informazioni relative alla query utente e ai documenti di supporto.

Come si è potuto evincere anche dagli esempi, la struttura del formato json in output ha un aspetto simile all'esempio in figura 2.5.

```
Examples of desired JSON output:
Instruct: "Mostrami tutti i pazienti nati dopo il 1940, visualizzando nome e data di nascita."
Output:
{{
"collection_name": "ANAGRAFICA",
"operation_type": "find",
"arguments": {{
    "filter": {{"DATADINASCITA": {{"$gt": "1930-01-01T00:00:00.000+00:00"}}}},
    "projection": {{"NOMEPAZ": 1, "DATADINASCITA": 1, "_id": 0}}
}}
}}

Instruct: "Quanti fumatori ci sono per ogni sezione, ordinati per sezione?"
Output:
{{
"collection_name": "ANAMNESI",
"operation_type": "aggregate",
"arguments": {{
    "pipeline": [
        {{"$match": {{"FUMO": "YES"}}}},
        {{"$group": {{"_id": "$SEZIONE", "count": {{"$sum": 1}}}}}},
        {{"$sort": {{"SEZIONE": -1}}}}
    ]
}}
}}
```

Figure 2.3: Seconda parte del system Prompt

```
The retrieved context is:
{context}

### Instruct (User's natural language query):
{user_instruction}

### Output (JSON object as a string):
"""
```

Figure 2.4: Terza parte del system Prompt

- **query_executor.py**: la classe all'interno di questo modulo, **MongoDBQueryExecutor** è la componente responsabile dell'esecuzione effettiva delle query MongoDB generate dal sistema.

  La funzione **execute_query** riceve un dizionario query_dict che incapsula tutti i dettagli della query (nome della collezione, tipo di operazione e argomenti). Supporta i due tipi di operazioni MongoDB più comuni per la lettura: **find** e **aggregate**.

  Dopo l'esecuzione della query, i risultati ottenuti da PyMongo possono contenere tipi di dati specifici di MongoDB (come ObjectId). La funzione _sanitize_data si occupa di convertire

Figure 2.5: Esempio di oggetto json creato dal generatore

questi tipi in formati serializzabili in JSON (ad esempio, gli ObjectId vengono convertiti in stringhe), rendendo i risultati immediatamente utilizzabili da altre parti del sistema o per la presentazione all'utente.

2.2.3 Package Evaluation

Questo package non è propriamente parte del sistema, ma è stato utilizzato in fase di valutazione per misurarne le performance. La metodologia verrà descritta più dettagliatamente in seguito, ma i moduli al suo interno sono i seguenti:

- **manual_query_executor.py**: modulo utilizzato per la creazione

e l'esecuzione delle query considerate **gold_standard**. I risultati di queste query verranno poi utilizzati per il confronto con quelle generate dall'agente.

- **evaluation.py**: modulo utilizzato per confrontare più file in formato csv. Si presume che i file abbiano almeno una colonna in comune che viene utilizzata per il confronto (di solito ID Univoco dei documenti). Restituisce i valori delle metriche di Precision, Recall, F1 Score e Jaccard Index.

2.2.4 Package Analytics

In questo package è presente un solo modulo dedito all'esecuzione di query predefinite e analisi dei dati per il recupero di informazioni cliniche specifiche

Il modulo **analytics_dashboard.py** è un insieme di funzioni che fanno proprio questo, ecco un elenco di query implementate nel modulo:

- **get_distribuzione_sesso**: Calcola la distribuzione dei pazienti per sesso.
- **get_distribuzione_comune_di_nascita**: ornisce i 20 comuni di nascita più frequenti tra i pazienti.
- **get_principali_cause_decesso**: Elenca le principali cause di decesso tra i pazienti

- **get_heart_failure_by_year**: Raggruppa i casi di insufficienza cardiaca per anno, fornendo una visione dell'andamento temporale.

Chapter 3

Benchmark e Valutazione

In questa sezione, vengono presentati i risultati delle performance del sistema, con e senza RAG, attraverso un set di query di difficoltà crescente.

3.1 Metodologia

Per validare in modo oggettivo l'efficacia della nostra applicazione, è stata definita una metodologia di benchmark ad-hoc per il nostro dataset.

3.1.1 Setup del Benchmark

Per la valutazione è stato quindi costruito un benchmark composto da 25 query in linguaggio naturale, suddivise per difficoltà in 3 categorie: **Easy**, **Medium** e **Difficult**.

- **Easy (10 query)**: Query che tipicamente richiedono una semplice operazione di `find` su una singola collezione, con filtri su campi diretti (es. *"restituiscimi tutti i pazienti nati a Teano"*).

- **Medium (10 query):** Query che possono richiedere aggregazioni semplici (`$group`), join tra due collezioni (`$lookup`) o filtri più articolati (es. *"conta quanti pazienti diabetici ci sono per ogni sezione"*).

- **Difficult (5 query)**: Query complesse che richiedono pipeline di aggregazione avanzate, con join multipli e calcoli complessi (*es. "per ogni sezione, trova il paziente con il BMI più alto e quello con il GFR più basso"*).

Per ogni query, è stata definita manualmente una *"Gold Standard Query"*, ovvero la query in pymongo corretta che rappresenta il risultato ideale.

Le performance del sistema sono state testate in due configurazioni: con e senza RAG.

3.1.2 Metriche di Valutazione

Sono state utilizzate le seguenti metriche standard per la valutazione del sistema, in maniera tale da quantificare l'accuratezza e la completezza dei risultati:

- **Precision**: Misura la **correttezza** dei risultati restituiti. Indica la proporzione di risultati pertinenti tra quelli recuperati ($P = \frac{TP}{TP+FP}$). È cruciale per garantire all'utente che i risultati mostrati siano affidabili.

- **Recall**: Misura la **completezza** dei risultati. Indica la proporzione di risultati pertinenti che il sistema è riuscito a recuperare sul totale dei risultati pertinenti esistenti ($R = \frac{TP}{TP+FN}$. Questa è fondamentale in ambito medico per assicurarsi di non perdere informazioni vitali.

- **F1 Score**: È la media armonica di Precisione e Recall ($F1 = 2 \cdot \frac{P \cdot R}{P+R}$). Fornisce un singolo valore che bilancia entrambe le metriche, ed è particolarmente utile quando sia la precisione che la completezza sono importanti.

- **Jaccard Index**: Misura la similarità tra l'insieme dei risultati recuperati e l'insieme dei risultati del Gold Standard ($J = \frac{TP}{TP+FP+FN}$). Fornisce una misura intuitiva di sovrapposizione tra risultati attesi e ottenuti.

3.2 Risultati

In questa sezione vengono presentati e analizzati i risultati ottenuti dal benchmark. L'analisi è suddivisa per livello di difficoltà per evidenziare in modo granulare i benefici dell'approccio RAG.

3.2.1 Analisi per livello di difficoltà

Query di difficoltà "Easy"

Metrica	Con RAG	Senza RAG
Precision	1.0000	1.0000
Recall	1.0000	1.0000
F1 Score	1.0000	1.0000
Jaccard	1.0000	1.0000

Table 3.1: Performance medie per le 10 query di difficoltà "Easy".

Come si evince dalla tabella, le metriche raggiungono tutte il valore di 1.0000 in entrambi gli scenari. Questo ci suggerisce che per le query più semplici, il sistema è in grado di generare risposte completamente accurate e pertinenti indipendentemente dal contesto aggiuntivo fornito da RAG. In pratica il Large language Model è già sufficientemente informato e non beneficia dell'arricchimento di contesto per task semplici.

Query di difficoltà "Medium"

Metrica	Con RAG	Senza RAG
Precision	0.7245	0.6031
Recall	0.8010	0.6010
F1 Score	0.7367	0.6015
Jaccard	0.7221	0.6007

Table 3.2: Performance medie per le 10 query di difficoltà "Medium".

Dalla tabella si evince già che il contesto RAG in questo caso ha reso il sistema è molto più efficace nel generare risposte accurate e

complete, rispetto alla sua controparte senza RAG. C'è da evidenziare comunque la non completa correttezza del modello in quanto per questo set di query si è raggiunto un risultato in tutte le metriche che sfiora il valore di 0.8

Query di difficoltà "Difficult"

Metrica	Con RAG	Senza RAG
Precision	0.9379	0.6978
Recall	1.0000	0.7978
F1 Score	0.9632	0.7311
Jaccard	0.9379	0.6957

Table 3.3: Performance medie per le 5 query di difficoltà "Difficult".

Il distacco fra le performance raggiunte dal modello con RAG e il modello senza RAG nel set di query "Difficult" è ancora più marcato. Questo risultato però possiamo attribuirlo anche in piccola parte al numero inferiore di query valutate in questa categoria. Pertanto si è deciso di effettuare anche una media pesata come valutazione finale.

3.2.2 Risultati complessivi

Premettiamo che questi risultati nonostante questi risultati non siano statisticamente significativi, possono dare un'idea di come il modello si comporti in questo determinato task con e senza la tecnologia RAG. Per concludere quindi il discorso, possiamo osservare dai risultati complessivi come questa tecnica migliora notevolmente la capacità del sis-

Metrica	Con RAG	Senza RAG
Precision	0.8774	0.7808
Recall	0.9204	0.7999
F1 Score	0.8873	0.7868
Jaccard	0.8764	0.7794

Table 3.4: Performance aggregate (Media Ponderata) su tutte le 25 query.

tema di comprendere e rispondere a una gamma più ampia e complessa di query. Sebbene per le query "Easy" le performance siano simili, l'apporto del RAG diventa cruciale per affrontare le query "Medium" e "Difficult", elevando significativamente l'accuratezza e la completezza delle risposte. Questo rende il sistema complessivamente più robusto e affidabile per l'interrogazione dei dati in linguaggio naturale.

Appendix A

Appendice

A.1 Risultati dettagliati del Benchmark

A.1.1 Query generate vs Gold Standard

Per motivi di leggibilità, si rimanda alla tabella presente nel pdf ***Benchmark_results.pdf*** nella repository github del progetto.

A.1.2 Risultati in dettaglio

ID Query	Difficoltà	Metrica	Con RAG	Senza RAG
1	Easy	**Precision**	1.0000	1.0000
		Recall	1.0000	1.0000
		F1 Score	1.0000	1.0000
		Jaccard	1.0000	1.0000
2	Easy	**Precision**	1.0000	1.0000
		Recall	1.0000	1.0000
		F1 Score	1.0000	1.0000
		Jaccard	1.0000	1.0000
3	Easy	**Precision**	1.0000	1.0000
		Recall	1.0000	1.0000
		F1 Score	1.0000	1.0000
		Jaccard	1.0000	1.0000
4	Easy	**Precision**	1.0000	1.0000
		Recall	1.0000	1.0000
		F1 Score	1.0000	1.0000
		Jaccard	1.0000	1.0000
5	Easy	**Precision**	1.0000	1.0000
		Recall	1.0000	1.0000
		F1 Score	1.0000	1.0000
		Jaccard	1.0000	1.0000

Table A.1: Risultati per le prime 5 query "Easy"

ID Query	Difficoltà	Metrica	Con RAG	Senza RAG
6	Easy	**Precision**	1.0000	1.0000
		Recall	1.0000	1.0000
		F1 Score	1.0000	1.0000
		Jaccard	1.0000	1.0000
7	Easy	**Precision**	1.0000	1.0000
		Recall	1.0000	1.0000
		F1 Score	1.0000	1.0000
		Jaccard	1.0000	1.0000
8	Easy	**Precision**	1.0000	1.0000
		Recall	1.0000	1.0000
		F1 Score	1.0000	1.0000
		Jaccard	1.0000	1.0000
9	Easy	**Precision**	1.0000	1.0000
		Recall	1.0000	1.0000
		F1 Score	1.0000	1.0000
		Jaccard	1.0000	1.0000
10	Easy	**Precision**	1.0000	1.0000
		Recall	1.0000	1.0000
		F1 Score	1.0000	1.0000
		Jaccard	1.0000	1.0000

Table A.2: Risultati per le ultime 5 query "Easy"

ID Query	Difficoltà	Metrica	Con RAG	Senza RAG
11	Medium	**Precision**	1.0000	1.0000
		Recall	1.0000	1.0000
		F1 Score	1.0000	1.0000
		Jaccard	1.0000	1.0000
12	Medium	**Precision**	1.0000	1.0000
		Recall	1.0000	1.0000
		F1 Score	1.0000	1.0000
		Jaccard	1.0000	1.0000
13	Medium	**Precision**	1.0000	1.0000
		Recall	1.0000	1.0000
		F1 Score	1.0000	1.0000
		Jaccard	1.0000	1.0000
14	Medium	**Precision**	0.0312	0.0312
		Recall	0.0100	0.0100
		F1 Score	0.0152	0.0152
		Jaccard	0.0076	0.0076
15	Medium	**Precision**	0.2138	1.0000
		Recall	1.0000	1.0000
		F1 Score	0.3522	1.0000
		Jaccard	0.2138	1.0000

Table A.3: Risultati per le prime 5 query "Medium"

ID Query	Difficoltà	Metrica	Con RAG	Senza RAG
16	Medium	**Precision**	0.0000	0.0000
		Recall	0.0000	0.0000
		F1 Score	0.0000	0.0000
		Jaccard	0.0000	0.0000
17	Medium	**Precision**	1.0000	0.0000
		Recall	1.0000	0.0000
		F1 Score	1.0000	0.0000
		Jaccard	1.0000	0.0000
18	Medium	**Precision**	1.0000	0.0000
		Recall	1.0000	0.0000
		F1 Score	1.0000	0.0000
		Jaccard	1.0000	0.0000
19	Medium	**Precision**	1.0000	1.0000
		Recall	1.0000	1.0000
		F1 Score	1.0000	1.0000
		Jaccard	1.0000	1.0000
20	Medium	**Precision**	1.0000	1.0000
		Recall	1.0000	1.0000
		F1 Score	1.0000	1.0000
		Jaccard	1.0000	1.0000

Table A.4: Risultati per le ultime 5 query "Medium"

ID Query	Difficoltà	Metrica	Con RAG	Senza RAG
21	Difficult	**Precision**	1.0000	0.0000
		Recall	1.0000	0.0000
		F1 Score	1.0000	0.0000
		Jaccard	1.0000	0.0000
22	Difficult	**Precision**	1.0000	1.0000
		Recall	1.0000	1.0000
		F1 Score	1.0000	1.0000
		Jaccard	1.0000	1.0000
23	Difficult	**Precision**	1.0000	1.0000
		Recall	1.0000	1.0000
		F1 Score	1.0000	1.0000
		Jaccard	1.0000	1.0000
24	Difficult	**Precision**	0.6897	0.5000
		Recall	1.0000	1.0000
		F1 Score	0.8163	0.6667
		Jaccard	0.6897	0.5000
25	Difficult	**Precision**	1.0000	0.9892
		Recall	1.0000	0.9892
		F1 Score	1.0000	0.9892
		Jaccard	1.0000	0.9786

Table A.5: Risultati per le query "Difficult"