



NLP Generative Task QnA

Relazione finale progetto Natural Language Processing

**Fabio di Gregorio
Ignazio Emanuele Piccichè
Lorenzo Pagliaricci**

Anno accademico 2024/2025

Indice

| | | |
|----------|---|-----------|
| 1 | Abstract | 2 |
| 2 | Introduzione | 3 |
| 3 | Metodo | 4 |
| 3.1 | MLX-LM Library | 4 |
| 3.1.1 | Introduzione MLX-LM | 4 |
| 3.1.2 | Implementazione di MLX-LM nel nostro progetto | 6 |
| 3.1.3 | Motivi per l'abbandono di MLX-LM | 7 |
| 3.2 | Transformers di Hugging Face | 8 |
| 3.3 | Sistema Retrieval-Augmented Generation (RAG) | 10 |
| 4 | Setup Sperimentale | 13 |
| 4.1 | Dataset e corpora | 13 |
| 4.1.1 | Divisione e preparazione dei dati | 14 |
| 4.1.2 | Preprocessing dei dati | 15 |
| 4.2 | Configurazione dei modelli per il fine-tuning | 18 |
| 4.2.1 | Instanziare modello, LoRA Adapter e Tokenizer | 18 |
| 4.2.2 | Parametri di training e pipeline di addestramento | 19 |
| 4.2.3 | Valutazione delle prestazioni: metriche custom | 20 |
| 4.3 | Configurazione del setup per il sistema RAG | 21 |
| 4.3.1 | Costruzione dell'indice vettoriale (<code>vector_store_impl.py</code>) | 21 |
| 4.3.2 | Implementazione del sistema RAG vero e proprio (<code>rag_impl.py</code>) | 22 |
| 4.3.3 | Note operative | 23 |
| 5 | Risultati e Conclusioni | 24 |
| 5.1 | Overview dei Risultati | 24 |
| 5.1.1 | Tabella delle Metriche | 24 |
| 5.1.2 | Approfondimento sulle performance di T5_SMALL_60M | 25 |
| 5.1.3 | Approfondimento sulle performance di TEXT_GEN_77M | 25 |
| 5.1.4 | Valutazione umana delle risposte e analisi qualitativa dei chunk recuperati | 26 |
| 5.1.5 | Conclusione | 27 |
| 5.2 | Limiti computazionali e problematiche riscontrate | 27 |
| 6 | Sviluppi Futuri | 29 |
| 6.1 | Potenziati sviluppi futuri | 29 |
| 6.1.1 | Utilizzo di modelli e strategie avanzate di retrieval | 29 |
| 6.1.2 | Espansione delle metriche di valutazione | 30 |
| 6.1.3 | Adattamento a nuovi domini | 30 |
| 6.1.4 | Ulteriori ottimizzazioni e automazione | 30 |

1 Abstract

In questo lavoro presentiamo un sistema di domanda-risposta generativo (RAG) sul dominio biomedico, progettato per affrontare i limiti di efficienza e generalizzazione dei modelli. A differenza delle soluzioni basate su dataset standard, abbiamo scelto **rag-mini-bioasq** per facilitare la sperimentazione con modelli leggeri e ottimizzati. Il problema centrale consiste nella selezione semantica e automatica dei contesti testuali rilevanti e nella generazione di risposte coerenti e fedeli al contesto stesso. Dopo una fase preliminare su Apple MLX, ci siamo orientati su **Transformers** (Hugging Face), adottando una pipeline articolata in tre fasi: retrieval semantico (**FAISS**), generazione tramite modelli sottoposti a fine-tuning con tecniche PEFT (*LoRA*), e valutazione automatica mediante metriche BLEU e Recall@k. L'infrastruttura, sviluppata interamente in Python, gestisce sia il retrieval che la generazione condizionata. I risultati mostrano che è possibile ottenere risposte accurate anche tramite modelli leggeri ed efficienti grazie a soluzioni di fine-tuning "parameter-efficient". Il codice sorgente del progetto è disponibile all'indirizzo: https://github.com/Lorenzo-Pagliaricci/NLP_Generative_Task_QnA.

Keywords: Retrieval-Augmented Generation, Biomedical NLP, Question Answering, FAISS, PEFT, LoRA, Hugging Face Transformers, Efficient Inference

2 Introduzione

Contesto e Motivazione L’elaborazione del linguaggio naturale (NLP) nel settore biomedico è una delle aree più cruciali e in rapida evoluzione della ricerca, in quanto la precisione informativa e la corretta interpretazione del linguaggio specialistico sono indispensabili per garantire affidabilità nei sistemi di question answering (QA). L’accumularsi esponenziale di dati testuali e la complessità concettuale tipica di questo dominio pongono continue sfide a modelli e metodiche tradizionali.

Stato dell’Arte e Lavori Correlati Negli ultimi anni, i grandi modelli linguistici (LLM) e i sistemi Retrieval-Augmented Generation (RAG) sono emersi come soluzioni di riferimento per il QA, specialmente in domini tecnici complessi. Approcci che combinano retrieval semantico e generazione condizionata si sono dimostrati particolarmente efficaci nel migliorare la precisione e la pertinenza delle risposte. Tuttavia, la maggior parte di questi sistemi richiede risorse computazionali rilevanti, limitando la diffusione e la scalabilità in contesti con vincoli di capacità.

Domande di Ricerca A fronte dei limiti sopra esposti, il nostro lavoro si pone le seguenti domande di ricerca:

- *È possibile ottenere risposte accurate ed efficienti in compiti di QA biomedico adottando modelli di dimensioni ridotte e strategie parameter-efficient?*
- *Quali sono i trade-off tra efficienza computazionale, qualità della generazione e scalabilità in questi contesti?*

Sintesi della Soluzione e Risultati Per rispondere a tali interrogativi, presentiamo una pipeline custom per il QA biomedico che integra retrieval semantico tramite FAISS e generazione condizionata basata su modelli perfezionati con tecniche PEFT (*LoRA*). L’architettura è pensata per lavorare efficacemente anche su risorse limitate (es. Google Colab, dispositivi Apple Silicon). I risultati preliminari evidenziano come sia possibile mantenere un buon livello di accuratezza, con vantaggi in termini di modularità, automazione e flessibilità di deployment. Ulteriori dettagli sul metodo, la valutazione e le prospettive future sono discussi nelle sezioni seguenti.

3 Metodo

Modelli Standard Come punto di partenza per il progetto, abbiamo analizzato le prestazioni di alcuni modelli preaddestrati non fine-tunati sul dominio biomedico, in modo da valutare la capacità intrinseca dei modelli generativi di produrre risposte coerenti e pertinenti senza alcuna specializzazione.

Abbiamo testato prevalentemente i seguenti modelli:

- hmbyt5/byt5-small-english
- ayushparwa12004/text-gen-v1-small
- Harshathemonster/t5-small-updated

Osservazioni iniziali (prima del fine tuning):

- I modelli non specificatamente fine-tunati per il dominio medico mostrano una comprensione limitata della terminologia specialistica, con risposte spesso generiche o vaghe.
- Pur essendo di norma corrette dal punto di vista grammaticale, le frasi generate risultano semanticamente povere e raramente approfondite rispetto alle domande specialistiche.
- Questi modelli sono stati quindi utilizzati come baseline al fine di quantificare i miglioramenti introdotti dalla fase di adattamento.

3.1 MLX-LM Library

3.1.1 Introduzione MLX-LM

Per il fine-tuning e l'inferenza dei modelli, abbiamo inizialmente esplorato la libreria **MLX-LM**¹, pensata per ottimizzare l'esecuzione di modelli AI, in particolare su dispositivi dotati di Apple Silicon (nel nostro caso, con processori M2 PRO ed M3). MLX-LM mira a combinare performance elevate con semplicità d'uso, consentendo l'esecuzione efficiente sia di modelli nativi MLX che di modelli convertiti da altri framework, grazie al suo backend a basso livello appositamente progettato per sfruttare le GPU/CPU Apple.

Caratteristiche principali:

- **Compatibilità con Hugging Face:** MLX-LM offre un'integrazione diretta con l'Hugging Face Hub², permettendo di scaricare e utilizzare migliaia di modelli LLM con un solo comando.

¹<https://github.com/ml-explore/mlx-lm>

²<https://huggingface.co/mlx-community>

- **Conversione e quantizzazione:** La libreria facilita la conversione di modelli da altri framework nel formato MLX, includendo la possibilità di quantizzarli per ridurre l'utilizzo di memoria e accelerare drasticamente l'esecuzione, mantenendo una qualità elevata.
- **Ottimizzazione hardware-specifica:** Il backend è progettato per sfruttare pienamente le capacità di calcolo delle CPU e GPU Apple Silicon, garantendo inferenza e fine-tuning rapidi anche su hardware consumer.
- **Supporto a fine-tuning e distribuzione:** MLX-LM consente sia il fine-tuning a basso costo di modelli completi o quantizzati, sia l'inferenza distribuita grazie al modulo `mx.distributed`.
- **Gestione di prompt e caching avanzato:** Supporta strumenti per la gestione efficiente di prompt molto lunghi, tramite cache versatili e key-value cache a dimensione fissa, ottimizzando il processamento di contesti estesi.

Vantaggi principali:

- **Efficienza su hardware Apple:** Grazie all'ottimizzazione per Apple Silicon, i modelli girano velocemente anche su macchine non dotate di GPU dedicate tradizionali, favorendo l'esecuzione locale, offline e a basso consumo.
- **Flessibilità nella gestione dei modelli:** L'utente può facilmente convertire, quantizzare e caricare nuovi modelli tramite CLI o API Python, anche con upload diretto su Hugging Face.
- **Facilità di utilizzo:** Comandi semplici per generazione, chat, e gestione avanzata di prompt, permettono una rapida prototipazione e sperimentazione.
- **Espandibilità:** Il supporto a una vasta gamma di modelli Hugging Face, insieme alla possibilità di aggiungere propri modelli convertiti, lo rende adatto sia per applicazioni di ricerca che di sviluppo leggero.

Limiti attuali:

- Al momento, MLX-LM non supporta completamente l'integrazione di modelli custom già fine-tunati con tecniche come LoRA, risultando meno adatto per workflow avanzati di personalizzazione.
- L'ecosistema e la documentazione sono ancora in sviluppo; alcune funzionalità avanzate sono limitate rispetto a framework più maturi.

3.1.2 Implementazione di MLX-LM nel nostro progetto

Tutti i file e risorse citate riguardanti MLX si possono trovare all'interno della cartella `MLX_LM` del progetto.

Nel nostro progetto la libreria `MLX-LM`³ è stata utilizzata per l'intero workflow di fine-tuning, fusione degli adapter e generazione delle risposte, in particolare su modelli leggeri ottimizzati per architetture Apple Silicon.

L'integrazione di `MLX-LM` è stata resa modulare e riproducibile tramite la scrittura di tre script principali (`fine_tune.sh`, `fuse.sh`, `generate.sh`), ciascuno dei quali automatizza una fase chiave della pipeline:

- **fine_tune.sh:** Script per l'avvio del fine-tuning parameter-efficient mediante LoRA. Utilizza il comando `mlx_lm.lora` per adattare un modello preaddestrato (ad es. `mlx-community/Phi-3-mini-128k-instruct-4bit`) su un dataset custom, specificando vari hyperparametri (batch size, learning rate, numero di layer, ecc.), il percorso per salvare gli adapter LoRA e tecniche di ottimizzazione della memoria come il gradient checkpointing.

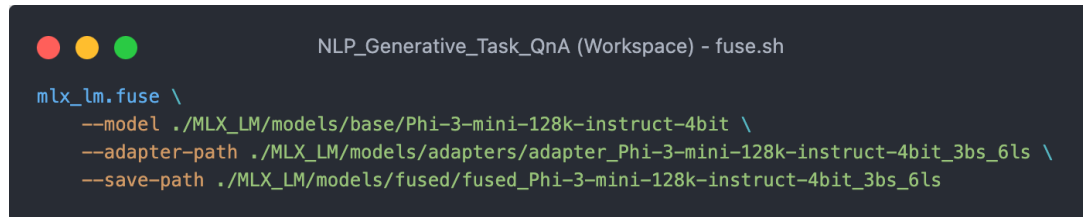


```
NLP_Generative_Task_QnA (Workspace) - fine_tune.sh

mlx_lm.lora \
  --model ./MLX_LM/models/base/Phi-3-mini-128k-instruct-4bit \
  --train \
  --adapter-path ./MLX_LM/models/adapters/adapter_Phi-3-mini-128k-instruct-4bit_3bs_6ls \
  --batch-size 3 \
  --val-batches 2 \
  --learning-rate 0.0001 \
  --num-layers 6 \
  --data MLX_LM/data/processed \
  --iters 200 \
  --steps-per-eval 50 \
  --seed 42 \
  --fine-tune-type lora \
  --grad-checkpoint \
  # --optimizer adam
```

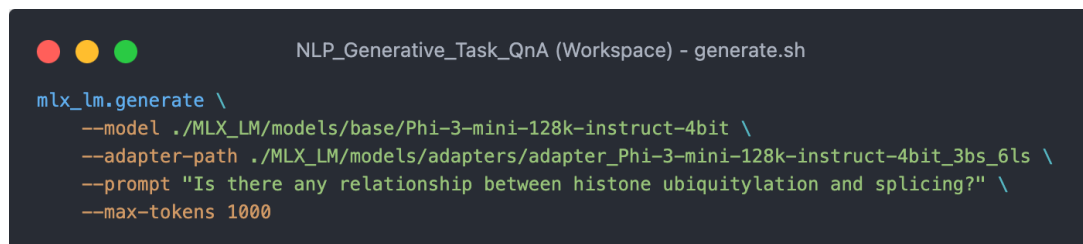
³<https://github.com/ml-explore/mlx-lm>

- **fuse.sh**: Script per la *fusione* dell'adattamento LoRA direttamente nei pesi del modello base. Tramite il comando `mlx_lm.fuse` si ottiene un modello stand-alone, pronto per l'inferenza anche senza necessità degli adapter separati.



```
NLP_Generative_Task_QnA (Workspace) - fuse.sh
mlx_lm.fuse \
  --model ./MLX_LM/models/base/Phi-3-mini-128k-instruct-4bit \
  --adapter-path ./MLX_LM/models/adapters/adapter_Phi-3-mini-128k-instruct-4bit_3bs_6ls \
  --save-path ./MLX_LM/models/fused/fused_Phi-3-mini-128k-instruct-4bit_3bs_6ls
```

- **generate.sh**: Script dedicato alla generazione di risposte testuali usando il modello (fuso o con adapter), su prompt custom e con controllo della lunghezza massima della generazione. È usato sia per valutare la qualità della generazione che per produrre output reali su domande biomediche.



```
NLP_Generative_Task_QnA (Workspace) - generate.sh
mlx_lm.generate \
  --model ./MLX_LM/models/base/Phi-3-mini-128k-instruct-4bit \
  --adapter-path ./MLX_LM/models/adapters/adapter_Phi-3-mini-128k-instruct-4bit_3bs_6ls \
  --prompt "Is there any relationship between histone ubiquitylation and splicing?" \
  --max-tokens 1000
```

Workflow complessivo:

1. *Fine-tuning LoRA*: avvio dell'adattamento parameter-efficient del modello di partenza usando script dedicati, con salvataggio degli adapter LoRA in un percorso personalizzato.
2. *Fusione LoRA*: per rendere il modello più leggero e facilmente utilizzabile, gli adapter vengono integrati direttamente nei pesi del modello.
3. *Generazione*: produzione e valutazione di risposte su prompt reali tramite script parametrizzabili, replicando un tipico scenario di domanda-risposta biomedico.

Questa struttura modulare, basata su script CLI standard di MLX-LM, garantisce riproducibilità, portabilità e semplicità di test sia su sistemi locali (Apple Silicon) che, all'occorrenza, su cluster distribuiti. Inoltre, l'uso dei parametri di quantizzazione e LoRA consente un'eccellente efficienza computazionale e rapidità anche su hardware consumer.

3.1.3 Motivi per l'abbandono di MLX-LM

Nonostante gli aspetti positivi della libreria MLX-LM, nel corso della sperimentazione abbiamo deciso di abbandonarne l'utilizzo come soluzione principale per il nostro progetto. Le ragioni principali riguardano una serie di limitazioni operative e di maturità dell'ecosistema:

- La libreria è ancora in una fase di sviluppo attivo: la documentazione è incompleta e la community di supporto limitata, rendendo difficile la risoluzione di problemi avanzati.
- Molti modelli di interesse per il nostro dominio non erano ottimizzati o ufficialmente supportati: la compatibilità risulta spesso parziale, riducendo la flessibilità nella scelta e sperimentazione dei modelli.
- Abbiamo riscontrato difficoltà nel fine-tuning su modelli custom o di dimensioni medio-grandi, con mancanza di strumenti e ottimizzazioni tipiche di framework più maturi.
- In generale, provando ad utilizzare modelli di dimensioni maggiori (o con architetture più recenti), abbiamo riscontrato notevoli rallentamenti sia in fase di training che di inferenza.

Per questi motivi, abbiamo deciso di orientarci verso framework più consolidati e flessibili, affrontando le problematiche di scalabilità e compatibilità in modo più ottimale, come sarà descritto nelle sezioni successive.

3.2 Transformers di Hugging Face

Tutti i file e risorse citate riguardanti l'implementazione Transformers si possono trovare all'interno della cartella `Transformers_Library` del progetto.

A seguito delle limitazioni riscontrate con MLX, abbiamo scelto come ambiente di riferimento la libreria `Transformers`⁴ di Hugging Face per tutte le fasi di fine-tuning, inferenza e validazione di modelli generativi. `Transformers` rappresenta lo stato dell'arte nell'NLP, grazie alla ricca collezione di modelli open source, pipeline integrate e la compatibilità con strumenti come `Datasets`, `PEFT`, e `Evaluate`.

Motivazioni principali della scelta:

- Ecosistema maturo, costantemente aggiornato, e vasta presenza di modelli preaddestrati per task e domini specialistici.
- Supporto completo e trasparente a strategie di parameter-efficient fine-tuning (LoRA, Adapters e altre tecniche PEFT).
- Facilità di integrazione, customizzazione e deployment, sia in locale che in cloud.

Prima della fase di training è stato realizzato un notebook dedicato al preprocessing e alla suddivisione del dataset nei tre split canonici (`train`, `validation`, `test`) (potete trovarlo in questo percorso nel progetto: *"Transformers_Library/prepare_datasetdict.ipynb"*).

Nelle fasi successive, l'implementazione operativa fa riferimento a tre file chiave, pensati per separare la pipeline di addestramento, la gestione degli adapter e la valutazione tramite metriche dedicate:

⁴<https://huggingface.co/docs/transformers/v4.51.3/index>

- **load_models.py**: questo script è il cuore della preparazione, addestramento e validazione del modello. Qui abbiamo:
 - Caricamento del modello e del tokenizer dalla libreria **transformers** tramite pipeline automatiche;
 - Configurazione dei parametri di addestramento (**Seq2SeqTrainingArguments**), inclusi batch size, learning rate, strategi di logging/salvataggio;
 - Integrazione del parameter-efficient fine-tuning tramite PEFT (LoRA) - qui si imposta la configurazione LoRA e la si applica al modello base;
 - Utilizzo delle metriche di valutazione ROUGE, BLEU, METEOR personalizzate attraverso **metrics_utils.py**, con funzioni specifiche per la decodifica delle sequenze e la gestione degli output predetti;
 - Salvataggio dei pesi dell'adapter e del tokenizer fine-tunato.
- **merge_base_and_adapt_ft_models.py**: una volta completato il fine-tuning tramite LoRA, questo script permette la fusione ("merge") degli adapter nel modello di base, generando un checkpoint finale stand-alone senza più dipendenza dagli adapter LoRA esterni. Sono previste gestione di errori, salvataggio del tokenizer e percorsi standardizzati per garantire la riutilizzabilità.
- **metrics_utils.py**: modulo di supporto dedicato al calcolo delle metriche di generazione (ROUGE, BLEU, METEOR), con pre-elaborazione di predizioni ed etichette (es: decodifica dei token e rimozione del padding), e output strutturato per integrare facilmente i risultati nei report sperimentali.

Sfide affrontate e misure adottate per la riproducibilità:

- Gestione della memoria e adattamento delle impostazioni (dimensione batch, gradient accumulation, precisione numerica) per lavorare su dispositivi con risorse limitate (GPU, Apple MPS).
- Attenta gestione delle versioni delle librerie **transformers**, **peft** e delle loro dipendenze: all'inizio di ogni script è stata inserita una stampa diagnostica delle versioni oppure sono state annotati i principali conflitti/requisiti nella documentazione del progetto, così da garantire la riproducibilità e facilitare il debugging.
- Strutturazione modulare dei file, salvataggio standardizzato dei checkpoint e adozione di pipeline di metriche custom (basate su **evaluate**) per poter effettuare confronti sistematici su diverse release e configurazioni delle librerie.
- Pur lavorando con modelli di dimensione contenuta, sono stati riscontrati rallentamenti in fase di training, richiedendo strategie aggiuntive per bilanciare tempo computazionale e performance ottenibile.

La combinazione di notebook iniziale per il preprocessing e degli script specialistici riportati offre quindi una pipeline trasparente, riproducibile e facilmente personalizzabile, consentendo confronti affidabili nella sperimentazione sia tra diversi modelli sia tra differenti versioni di **transformers**, **peft** e ambiente di calcolo.

3.3 Sistema Retrieval-Augmented Generation (RAG)

Tutti i file e risorse citate riguardanti l'implementazione del sistema RAG si possono trovare all'interno della cartella `RAG_system` del progetto.

Per incrementare l'accuratezza e la specificità delle risposte nel dominio biomedico, è stato implementato un sistema **Retrieval-Augmented Generation (RAG)**. Questo approccio integra la potenza dei modelli generativi con un modulo di recupero semantico che consente di fornire informazioni specifiche recuperate dinamicamente dal corpus di riferimento.

Architettura del sistema:

- **Retrieval:** recupero automatico dei chunk di testo più pertinenti rispetto alla domanda dell'utente, utilizzando un indice FAISS costruito su embedding semantici (`BAAI/bge-base-en-v1.5`).
- **Generazione:** produzione di risposte guidata esclusivamente dal contesto recuperato tramite un modello generativo fine-tunato, il cui input include il *prompt di sistema* ed i *chunk* selezionati come *contesto*.

Scelta del modello di embedding Per la componente di recupero semantico del sistema RAG, la selezione del modello di embedding rappresentava un aspetto cruciale, poiché la qualità degli embedding influenza direttamente la capacità di identificare i chunk più rilevanti rispetto alle query utente. A tal fine, è stata condotta un'analisi comparativa dei principali modelli disponibili su Hugging Face, tenendo come riferimento i benchmark e i risultati riportati nella **MTEB Leaderboard**⁵, piattaforma di riferimento per la valutazione delle performance dei modelli di embedding su una varietà di task tra cui search, retrieval e semantic similarity.

Dall'analisi emerge come il modello `BAAI/bge-base-en-v1.5` risulti regolarmente tra i più performanti secondo la MTEB Leaderboard, in particolare sui task di retrieval testuale in lingua inglese. Il modello si distingue per robustezza cross-domain, efficienza computazionale e facilità di integrazione con i principali framework Python. Queste caratteristiche, unite ai risultati sperimentali pubblici, ci hanno portato a preferirlo rispetto ad alternative meno performanti o con requisiti computazionali superiori, considerato anche il nostro focus sul dominio biomedicale dove la precisione semantica è essenziale.

Pertanto, sulla base della comparazione oggettiva proposta da MTEB Leaderboard e delle nostre esigenze applicative, abbiamo adottato `BAAI/bge-base-en-v1.5` come backbone per la generazione degli embedding all'interno del sistema RAG.

⁵<https://huggingface.co/spaces/mteb/leaderboard>

L'intero sistema è stato implementato utilizzando due file fondamentali, ciascuno con responsabilità specifiche:

Creazione dell'indice vettoriale - `vector_store_impl.py`

Il file `vector_store_impl.py` contiene il codice utilizzato per trasformare un dataset testuale in un indice vettoriale FAISS⁶, che costituisce il database interrogabile dal sistema di retrieval. Per la scelta del *Vector store* ci siamo studiati la documentazione sui *Vector stores di LangChain*⁷. L'obiettivo principale è pre-elaborare il corpus in modo efficiente e strutturato.

- **Caricamento del dataset:** viene scaricato e caricato un dataset predefinito (`enelpol/rag-mini-bioasq`) dalla piattaforma Hugging Face, con la creazione di oggetti `Document` che associano i passaggi testuali a metadati utili per il recupero.
- **Chunking dei documenti:** i documenti vengono divisi in chunk di testo più piccoli utilizzando `RecursiveCharacterTextSplitter`. Questo garantisce una migliore granularità nella ricerca dei chunk semantici più pertinenti.
- **Generazione dell'indice FAISS:** ogni chunk è convertito in un vettore numerico tramite un modello di embedding (`BAAI/bge-base-en-v1.5`) e successivamente indicizzato con FAISS per consentire il retrieval.
- **Salvataggio dell'indice:** l'indice FAISS risultante viene salvato su disco per un utilizzo successivo nelle fasi di retrieval.

Flusso generale:

```
# Creazione dell'indice FAISS
vector_store = FAISS.from_documents(chunked_documents, embeddings)
vector_store.save_local("faiss_index_bioasq")
```

Sistema RAG - `rag_impl.py`

Il file `rag_impl.py` rappresenta l'applicazione del sistema Retrieval-Augmented Generation, orchestrando le fasi di retrieval e generazione.

- **Caricamento degli strumenti:** sono caricati il modello di embedding (`BAAI/bge-base-en-v1.5`) e l'indice FAISS pre-costruito.
- **Retrieval semantico:** a partire dalla query dell'utente, questa viene trasformata in un embedding e confrontata con i vettori presenti nell'indice FAISS. Si recuperano i k chunk più simili utilizzando una ricerca per similarità coseno.
- **Preparazione del contesto:** i chunk recuperati vengono concatenati e combinati con il prompt del sistema per creare l'input da fornire al modello generativo.

⁶<https://python.langchain.com/docs/integrations/vectorstores/faiss/>

⁷<https://python.langchain.com/docs/integrations/vectorstores/>

- **Generazione della risposta:** utilizzando un modello generativo Transformers fine-tunato, viene prodotta una risposta condizionata esclusivamente sul contesto fornito.
- **Valutazione delle metriche:** il sistema calcola metriche come ROUGE, BLEU e METEOR confrontando le risposte generate con quelle di riferimento, grazie alle utility di `metrics_utils.py`.

Flusso generale:

```
# Retrieval dei chunk più rilevanti
retrieve = vector_store.similarity_search_with_score_by_vector(
    embedding=embeddings.embed_query(query), k=15)
# Generazione della risposta
output = text_generator(prompt_text, max_length=1000)
```

Sfide affrontate:

- *Ottimizzazione delle query:* la definizione di embedding e parametri per il retrieval ha richiesto tuning e iterazioni per migliorare la rilevanza dei chunk recuperati.
- *Bilanciamento memoria-prestazioni:* la gestione degli indici FAISS e del calcolo degli embedding è stata ottimizzata per sfruttare dispositivi MPS (Apple Silicon).
- *Valutazione accurata:* le metriche di qualità RAG (ROUGE, BLEU, METEOR) sono state personalizzate per rispondere agli obiettivi dominanti del dominio bio-medico.

Conclusione: L'integrazione di un sistema RAG ha permesso di migliorare significativamente l'accuratezza e la specificità delle risposte, supportando il modello generativo con recupero semantico dinamico.

Per i dettagli tecnici si rimanda al capitolo successivo, *Setup Sperimentale*.

4 Setup Sperimentale

4.1 Dataset e corpora

Per l'intera fase sperimentale è stato utilizzato il dataset `rag-mini-bioasq`, pubblicamente disponibile su Hugging Face.

Questa versione del dataset, rispetto a quella fornita dalla traccia, è stata scelta perché, grazie a una serie di modifiche e processi di pulizia, risulta più pulita, coerente e organizzata rispetto ai dataset grezzi o meno strutturati normalmente disponibili per il question answering biomedico.

In particolare, le principali modifiche apportate includono:

- **Riempimento dei passaggi mancanti:** i passaggi che originariamente contenevano valori "nan" sono stati corretti con i testi effettivi mancanti.
- **Correzione del tipo di `relevant_passage_ids`:** il campo `relevant_passage_ids` nelle triplette QAP è stato convertito da stringa a sequenza di interi, assicurando maggiore uniformità nei riferimenti tra domande e passaggi.
- **Deduplicazione dei passaggi:** sono stati rimossi 40 passaggi duplicati. Di conseguenza, tutti i riferimenti (`relevant_passage_ids`) nelle triplette QAP sono stati aggiornati per puntare correttamente agli identificativi dei passaggi deduplicati.
- **Suddivisione in train/test:** le triplette QAP sono state suddivise in due insiemi distinti, di addestramento e di test, per consentire una valutazione quantitativa chiara e strutturata delle performance dei modelli.

Queste modifiche rendono il dataset `rag-mini-bioasq` particolarmente adatto a esperimenti di modelling e valutazione nel campo del question answering biomedico, grazie a una maggiore affidabilità e pulizia dei dati di partenza.

Caratteristiche principali:

- **Lingua:** inglese.
- **Formato:** JSON, contenente triplette strutturate (`question`, `answer`, `passage`).
- **Campi:**
 - `question`: testo della domanda posta.
 - `answer`: risposta corretta (target).
 - `passage`: blocchi di testo da cui deriva o può essere estratta la risposta.
 - `relevant_passage_ids`: lista degli ID degli specifici passaggi considerati rilevanti per la domanda.

Il dataset `rag-mini-bioasq` è stato scelto poiché, rispetto ad alternative più estese e complesse, si presenta compatto, facile da processare ed è già ottimizzato per la costruzione di pipeline RAG, rendendo più agevole sia l'elaborazione che l'integrazione con modelli più leggeri o sperimentali.

4.1.1 Divisione e preparazione dei dati

Una volta scaricato, il dataset è stato suddiviso in tre partizioni distinte:

- **Train:** utilizzato per il fine-tuning effettivo dei modelli;
- **Validation:** impiegato per valutazioni intermedie, scelta degli iperparametri e strategie di early stopping;
- **Test:** riservato alla valutazione finale e al confronto sistematico tra approcci.

La divisione è stata effettuata tramite notebook dedicato, che potete trovare in questo percorso del progetto "*Transformers_Library/prepare_datasetdict.ipynb*", sfruttando funzioni fornite dalla libreria `datasets` di Hugging Face.

4.1.2 Preprocessing dei dati

Per garantire la coerenza tra il task di question answering biomedico e la pipeline di addestramento, il preprocessing viene applicato dinamicamente (“on-the-fly”) tramite una funzione dedicata, come mostrato negli snippet di codice in Figura 1.

Il cuore della procedura consiste nell’arricchimento della domanda con un **prompt di sistema**, appositamente formulato per il contesto biomedico. Prima di essere tokenizzata, ciascuna domanda viene infatti preceduta dalla seguente istruzione di sistema:

```
You are a helpful reading assistant who answers questions. Be concise.  
If you're unsure, just say that you don't know. \n\nQuestion:
```

Questo approccio permette di inquadrare con chiarezza l’attesa semantica per il modello generativo, riducendo ambiguità e favorendo risposte brevi, aderenti e precise.

La funzione di preprocessing esegue quindi i seguenti passaggi fondamentali:

- **Combinazione del prompt con la domanda:** la domanda originale viene concatenata al prompt specializzato, andando a formare la sequenza di input finale per il modello.
- **Tokenizzazione:** sia la sequenza di input (prompt + domanda) che la risposta di riferimento (*target*) sono tokenizzate separatamente, con gestione di padding e troncamento per adattarsi alla lunghezza massima consentita dal modello (fino a 512 token).
- **Preparazione dei campi di training:** vengono generati e restituiti tutti i campi richiesti per il fine-tuning, ovvero `input_ids` e `attention_mask` per l’input, oltre a `labels` per la supervisione della generazione.
- **Pulizia delle colonne non necessarie:** una volta completata la tokenizzazione, tutte le colonne originarie come `question`, `answer`, `id`, e `relevant_passage_ids` vengono rimosse per lasciare solo i tensori utili all’addestramento (Figura 1).


```
NLP_Generative_Task_QnA - prepare_datasetdict.ipynb

# Define the preprocessing function
def preprocess_function(examples):
    # Define the system prompt to be prepended to questions
    FINETUNING_SYSTEM_PROMPT = """You are a helpful reading assistant who answers questions.
    Be concise. If you're unsure, just say that you don't know. \n\nQuestion: """
    inputs = [FINETUNING_SYSTEM_PROMPT + q for q in examples["question"]]

    # Tokenize the inputs (questions) with padding and truncation
    model_inputs = tokenizer(
        inputs,
        max_length=512,
        truncation=True,
        # padding="longest"
    )

    # Tokenize the targets (answers) to create the labels
    labels = tokenizer(
        text_target=examples["answer"],
        max_length=512,
        truncation=True,
        # padding="max_length",
    )

    # Add the labels to the model inputs
    model_inputs["labels"] = labels["input_ids"]

    return model_inputs

NLP_Generative_Task_QnA - prepare_datasetdict.ipynb

# Apply the preprocessing function to the dataset
tokenized_datasets = dataset_dict.map(preprocess_function, batched=True)

# Remove the original columns as they are no longer needed by the model
tokenized_datasets = tokenized_datasets.remove_columns(
    ["question", "answer", "id", "relevant_passage_ids"]
)
```

Figura 1: Esempi di preprocessing: applicazione della funzione di tokenizzazione arricchita con prompt per la pipeline biomedica.

Questo flusso assicura che ogni dato in ingresso sia interpretato dal modello coerentemente con il task richiesto, ottimizzando il processo di apprendimento supervisionato per la generazione di risposte accurate nel dominio biomedico.

Questa strategia rende il workflow adattabile sia a dataset già tokenizzati sia a dati "grezzi", garantendo standardizzazione e massima riproducibilità, senza perdere flessibilità nell'integrazione di prompt specifici o nella scelta del tokenizer più adeguato al modello usato.

4.2 Configurazione dei modelli per il fine-tuning

La configurazione dei modelli per il fine-tuning, ispirata e implementata nello script `load_models.py`, è stata realizzata secondo pipeline riproducibili ed adattabili, con attenzione sia alla preparazione del modello che all'impostazione dei criteri di valutazione.

4.2.1 Istanziare modello, LoRA Adapter e Tokenizer

L'inizializzazione del modello di sequenza-to-sequenza (Seq2Seq) con quantizzazione e parameter-efficient fine-tuning (LoRA) avviene tramite la libreria `transformers` e `peft`:

```
NLP_Generative_Task_QnA - load_models.py

# --- Load Models ---
# Get the model name/path from the loaded configuration
MODEL_NAME = "BYT5_SMALL_300M"
MODEL = config["BYT5_SMALL_300M"]
PREPARED_DATASET = config.get("TOKENIZED_DATASET", config["PREPARED_DATASET"])
SAVED_MODEL_PATH = config["SAVED_MODEL_PATH"]

# Define the quantization configuration using TorchAoConfig for int8 weight-only quantization
quantization_config = TorchAoConfig("int8_weight_only")

config = AutoConfig.from_pretrained(
    MODEL,
    torch_dtype=torch.bfloat16, # Use bfloat16 for mixed-precision inference
    torch_dtype=torch.float32, # Use float32 for mixed-precision inference
    device_map="auto", # Map the model to CPU (or "auto" for automatic mapping)
    # NOTE: non è la quantizzazione il problema, riabilitarla
    quantization_config=quantization_config, # Apply the defined quantization configuration
)

# Load the pre-trained Seq2Seq language model (T5)
model = AutoModelForSeq2SeqLM.from_config(
    config, # Load the model configuration
)

# --- LoRA Configuration ---
# Define the LoRA configuration for model adaptation
lora_config = LoraConfig(
    task_type=TaskType.SEQ_2_SEQ_LM, # type of task to train on
    inference_mode=False, # set to False for training
    r=2, # dimension of the smaller matrices
    lora_alpha=40, # scaling factor
    lora_dropout=0.5, # dropout of LoRA layers
    target_modules=[
        "k",
        "v",
        "q",
        "o",
    ], # Specify the target modules for LoRA adaptation (Updated for M2M100)
)

# Apply LoRA using get_peft_model
model = get_peft_model(model, lora_config)

# model.gradient_checkpointing_enable() # Enable gradient checkpointing to save memory during training

# Load the tokenizer associated with the specific T5 model variant being used
tokenizer = AutoTokenizer.from_pretrained(MODEL)
```

4.2.2 Parametri di training e pipeline di addestramento

I parametri di training vengono stabiliti tramite la classe `Seq2SeqTrainingArguments` di Hugging Face, che consente il salvataggio custom degli output, setting batch size, logging, strategie di early stopping, ecc.:

```
NLP_Generative_Task_QnA - load_models.py

# --- Load Training Arguments ---
training_args = Seq2SeqTrainingArguments(
    output_dir="Transformers_Library/results", # Directory to save the model and training outputs
    per_device_train_batch_size=5, # Batch size for training on each device
    per_device_eval_batch_size=5, # Batch size for evaluation on each device
    gradient_accumulation_steps=5, # Number of steps to accumulate gradients before updating weights
    num_train_epochs=10, # Total number of training epochs
    logging_dir=f"Transformers_Library/tensorboard_logs/{MODEL_NAME}", # Directory for storing logs
    logging_steps=10, # Log every 10 steps
    save_steps=50, # Save the model every 500 steps
    eval_strategy="steps", # Evaluate the model every 'eval_steps'
    do_eval=True, # Perform evaluation during training
    eval_steps=50, # Evaluate every 500 steps
    seed=42, # Random seed for reproducibility
    load_best_model_at_end=True, # Load the best model at the end of training
    metric_for_best_model="eval_loss", # Metric to determine the best model
    greater_is_better=False, # Whether a higher metric value is better
    learning_rate=1e-5, # Learning rate
    warmup_steps=250, # Number of warmup steps for learning rate scheduler
    data_loader_num_workers=0, # Number of subprocesses to use for data loading
    eval_accumulation_steps=25, # Muove ogni batch subito in CPU, evitando di creare buffer grandi
    eval_on_start=False, # Evaluate at the start of training
    predict_with_generate=True, # Necessary for Seq2Seq tasks to generate predictions
)
```

Esempio di inizializzazione Trainer:

```
NLP_Generative_Task_QnA - load_models.py

# Create a data collator for padding sequences in batches efficiently
data_collator = DataCollatorForSeq2Seq(
    tokenizer=tokenizer,
    model=model,
    # padding="max_length",
    padding="longest",
)
```

```

NLP_Generative_Task_QnA - load_models.py

trainer = Seq2SeqTrainer(
    model=model,
    args=training_args,
    train_dataset=dataset["train"],
    eval_dataset=dataset["validation"], # .select(range(100)),
    data_collator=data_collator,
    compute_metrics=compute_metrics_for_trainer, # Use the wrapper function
    tokenizer=tokenizer, # Pass the tokenizer to the Trainer
)

```

4.2.3 Valutazione delle prestazioni: metriche custom

Per la valutazione delle prestazioni sono state progettate funzioni custom (vedi `Transformers.Library/metrics_utils.py`) che decodificano i token, rimuovono il padding e calcolano ROUGE, BLEU e METEOR, integrando la pipeline `evaluate` di Hugging Face:

```

metric_rouge = load("rouge")
metric_bleu = load("bleu")
metric_meteor = load("meteor")

def compute_metrics_for_trainer(eval_preds):
    preds, labels = eval_preds
    return compute_metrics_base(
        preds, labels, tokenizer, metric_rouge, \
        metric_bleu, metric_meteor
    )

```

Esempio di funzione di base:

```

NLP_Generative_Task_QnA - load_models.py

def compute_metrics_base(predictions, labels, tokenizer, metric_rouge, metric_bleu, metric_meteor):
    labels = np.where(labels != -100, labels, tokenizer.pad_token_id)
    decoded_preds = tokenizer.batch_decode(predictions, skip_special_tokens=True)
    decoded_labels = tokenizer.batch_decode(labels, skip_special_tokens=True)
    results = {}
    rouge_output = metric_rouge.compute(predictions=decoded_preds, references=decoded_labels)
    bleu_output = metric_bleu.compute(predictions=decoded_preds, references=decoded_labels)
    meteor_output = metric_meteor.compute(predictions=decoded_preds, references=decoded_labels)
    results.update(rouge_output)
    results.update(bleu_output)
    results.update(meteor_output)
    return results

```

Salvataggi e reproducibilità: Al termine dell'addestramento vengono salvati pesi, configurazioni e tokenizer per garantire la reproducibilità dei risultati.

```
NLP_Generative_Task_QnA - load_models.py

# Save the trained model
trainer.save_model(
    SAVED_MODEL_PATH + "_" + MODEL_NAME + '_V2' # Use MODEL_NAME for consistency
) # Save the model to the specified directory
# Save the tokenizer
tokenizer.save_pretrained(
    SAVED_MODEL_PATH
    + "_"
    + MODEL_NAME
    + "_"
    + "Tokenizer" # Use MODEL_NAME for consistency
    + '_V2'
) # Save the tokenizer to the same directory
```

Nota: La modularità del codice permette sia la tokenizzazione dinamica (on-the-fly) sia il caricamento diretto di dataset già pronti, adattando così la pipeline a varie esigenze sperimentali.

4.3 Configurazione del setup per il sistema RAG

L'implementazione del sistema RAG (Retrieval-Augmented Generation) è articolata in due fasi principali, ciascuna gestita da uno script dedicato: `RAG_system/vector_store_impl.py` per la costruzione dell'indice vettoriale, e `RAG_system/rag_impl.py` per l'orchestrazione del retrieval e generazione.

4.3.1 Costruzione dell'indice vettoriale (`vector_store_impl.py`)

Il primo passaggio consiste nella creazione di un database vettoriale interrogabile (indice FAISS). Le fasi operative del file sono:

- **Caricamento del dataset:** il dataset `enelpol/rag-mini-bioasq` viene importato da Hugging Face tramite `load_dataset()`, con selezione dello split desiderato.
- **Preparazione dei documenti:** ogni esempio viene trasformato in un oggetto `Document` di LangChain, strutturato con il contenuto testuale e i relativi metadati (ID, source).
- **Chunking:** i passaggi vengono suddivisi in unità più piccole tramite `RecursiveCharacterTextSplitter` per facilitare la granularità del retrieval.

- **Embedding:** per ogni chunk viene calcolato l'embedding numerico tramite il modello BAAI/bge-base-en-v1.5, selezionato per elevate performance su benchmark di semantic retrieval.
- **Creazione batch e indicizzazione:** i chunk vengono processati in batch per ottimizzazione della memoria, creando infine la struttura FAISS (`FAISS.from_documents()`), che viene salvata localmente su disco per utilizzo futuro.

Esempio (dal codice):

```

1 dataset = load_dataset("enelpol/rag-mini-bioasq", "text-corpus")
2 documents = [Document(page_content=doc["passage"], ... ) for doc
3               in dataset["test"]]
4 chunked_documents = RecursiveCharacterTextSplitter(chunk_size
5               =200, chunk_overlap=20).split_documents(documents)
6 embeddings = HuggingFaceEmbeddings(model_name="BAAI/bge-base-en-
  v1.5")
7 vector_store = FAISS.from_documents(first_batch, embeddings)
8 vector_store.save_local("faiss_index_bioasq")

```

4.3.2 Implementazione del sistema RAG vero e proprio (rag_impl.py)

Il secondo script gestisce l'intero flusso retrieval+generazione, integrando tutte le componenti operative:

- **Caricamento embedding e indice:** viene ricaricato il modello di embedding e l'indice FAISS salvato nella fase precedente.
- **Gestione query utente:** la domanda viene fornita in input al sistema.
- **Retrieval semantico:** la query viene convertita in embedding; grazie alla funzione `similarity_search_with_score_by_vector()` si effettua una ricerca semantica all'interno dell'indice FAISS per recuperare i chunk più pertinenti.
- **Costruzione del prompt:** i chunk recuperati vengono concatenati a un system prompt, utilizzato come input contestuale per il modello generativo.
- **Generazione della risposta:** viene utilizzata una pipeline Hugging Face locale tramite `transformers.pipeline()`, ospitata su acceleratore hardware disponibile, che fornisce la risposta in linguaggio naturale in base ai chunk.
- **Valutazione:** sono implementate metriche custom (ROUGE, BLEU, METEOR) per la valutazione quantitativa delle risposte generate, tramite `evaluate` e funzioni dedicate.

Esempio (dal codice):

```
1 embeddings = HuggingFaceEmbeddings(model_name="BAAI/bge-base-en-  
  v1.5")  
2 vector_store = FAISS.load_local("faiss_index_bioasq", embeddings,  
  ...)  
3 retrieve = vector_store.similarity_search_with_score_by_vector(  
4   embedding=embeddings.embed_query(query), k=15)  
5 context_for_llm = "\n".join([chunk.page_content for chunk, _ in  
  most_similar_chunks])  
6 output = text_generator(prompt_text, max_length=1000)
```

4.3.3 Note operative

Nel setup:

- È fondamentale utilizzare lo stesso modello di embedding sia per la creazione dell'indice che per l'inferenza.
- La pipeline di retrieval tramite FAISS consente tempi di risposta rapidi anche su dataset medio-grandi.
- Tutti i parametri (dimensioni chunk, batch size, k chunk top- k recuperati, ecc.) sono configurabili in funzione delle risorse computazionali disponibili.

5 Risultati e Conclusioni

5.1 Overview dei Risultati

In questa sezione vengono presentati e analizzati i risultati sperimentali ottenuti dal sistema di question answering generativo. L’analisi si focalizza su un confronto sistematico tra i diversi modelli testati, con particolare attenzione alle differenze tra i modelli di base e quelli fine-tunati. Le performance sono valutate secondo due dimensioni principali:

- **Metriche di generazione:** includono *ROUGE*, *METEOR* e *BLEU*, calcolate tramite le procedure standardizzate sviluppate nei nostri script (`metrics_utils.py`). Queste metriche misurano la coerenza, la correttezza semantica e la sovrapposizione tra le risposte generate dal modello e quelle di riferimento.
- **Metriche di retrieval:** ove applicabile (sistemi RAG), sono considerate misure come *Recall@k*, che valutano la capacità del sistema di recuperare informazioni pertinenti dal database vettoriale in fase di generazione della risposta.

La presentazione dei risultati, comprensiva di tabelle comparative e sintesi qualitative, permette di evidenziare l’impatto dei diversi modelli e approcci di training sulle prestazioni finali. In particolare, sarà possibile osservare le differenze tra modelli di taglia e architettura differenti (ad esempio `T5_SMALL_60M`, `TEXT_GEN_77M`, `BYT5_SMALL_300M`), mettendo in luce punti di forza e limiti emersi dalle valutazioni. Tale analisi quantitativa e qualitativa guida la discussione sulle scelte architetturali e sulle prospettive di miglioramento future.

5.1.1 Tabella delle Metriche

| Modello | ROUGE-1 | ROUGE-2 | ROUGE-L | ROUGE-Lsum | BLEU | METEOR |
|-----------------|---------------|---------------|---------------|---------------|---------------|---------------|
| T5_SMALL_60M | 0.2383 | 0.0916 | 0.1857 | 0.1871 | 0.0769 | 0.2362 |
| BYT5_SMALL_300M | 0.0729 | 0.0148 | 0.0582 | 0.0581 | 0.0074 | 0.0655 |
| TEXT_GEN_77M | 0.1447 | 0.0633 | 0.1293 | 0.1305 | 0.0050 | 0.0867 |

Tabella 1: Metriche di Generazione sui Tre Modelli Testati (Test Set)

| Modello | Precisioni | Brevity Penalty | Length Ratio | Translation Length | Reference Length |
|-----------------|---|-----------------|---------------|--------------------|------------------|
| T5_SMALL_60M | [0.2371, 0.0808, 0.0499, 0.0366] | 1.0000 | 1.1386 | 31376 | 27556 |
| BYT5_SMALL_300M | [0.0783, 0.0110, 0.0032, 0.0011] | 1.0000 | 1.1904 | 32802 | 27556 |
| TEXT_GEN_77M | [0.3563, 0.1633, 0.1182, 0.0967] | 0.0313 | 0.2241 | 6174 | 27556 |

Tabella 2: Altre Metriche Calcolate sui Tre Modelli Testati (Test Set)

Legenda modelli:

- `T5_SMALL_60M` → Harshathemonster/t5-small-updated
- `BYT5_SMALL_300M` → hmbyt5/byt5-small-english

- **TEXT_GEN_77M** → `ayushparwal2004/text-gen-v1-small`

ROUGE-1, *ROUGE-2*, *ROUGE-L*, *ROUGE-Lsum* misurano la sovrapposizione tra n-gram/sequenze generate e di riferimento; *BLEU* valuta la precisione n-gram; *METEOR* considera la corrispondenza semantica e lessicale.

Sintesi dei risultati:

Dall’analisi delle metriche emerge che **T5_SMALL_60M** (`Harshathemonster/t5-small-updated`) si distingue come il modello con le migliori performance generali su tutti gli indicatori (*ROUGE*, *BLEU* e *METEOR*), presentandosi come soluzione preferibile per risposte accurate e coerenti. **BYT5_SMALL_300M** (`hmbyt5/byt5-small-english`) risulta il più debole tra i tre modelli valutati, mentre **TEXT_GEN_77M** (`ayushparwal2004/text-gen-v1-small`) mostra prestazioni intermedie. Questi dati guidano nella selezione del modello più adatto ai requisiti applicativi del task considerato.

5.1.2 Approfondimento sulle performance di T5_SMALL_60M

Nei nostri esperimenti, il modello **T5_SMALL_60M** (`Harshathemonster/t5-small-updated`), pur essendo il più compatto tra i testati, ha ottenuto le migliori performance secondo tutte le metriche di riferimento (*ROUGE*, *BLEU*, *METEOR*). Questo risultato, solo apparentemente controintuitivo, può essere attribuito a diversi fattori specifici:

- **Ottimo bilanciamento tra capacità e generalizzazione:** Un modello di dimensioni contenute, se ben configurato, è meno soggetto a fenomeni di overfitting, specialmente quando le dimensioni del dataset di fine-tuning sono limitate o molto focalizzate su un dominio specifico.
- **Adattamento più efficace al task:** Il checkpoint usato potrebbe essere stato pre-addestrato o adattato su dati più simili a quelli biomedici presenti nel nostro dataset, favorendo una migliore “affinità” semantica tra input e output.
- **Efficienza computazionale:** I modelli piccoli risultano più stabili durante addestramento ed inferenza (soprattutto su hardware consumer), consentendo interpretazione ed ottimizzazione degli hyperparametri più mirata ed efficace rispetto ai modelli più grandi.
- **Output più focalizzati e meno rumorosi:** Dai risultati delle metriche emerge che **T5_SMALL_60M** tende a produrre risposte concise e aderenti alla domanda, mentre i modelli più grandi possono cadere in generazioni ridondanti o fuori tema.

5.1.3 Approfondimento sulle performance di TEXT_GEN_77M

Nonostante il modello **T5_SMALL_60M** sia risultato il più performante sulle metriche globali di generazione (*ROUGE*, *BLEU*, *METEOR*), l’analisi delle metriche supplementari rivela che **TEXT_GEN_77M** (`ayushparwal2004/text-gen-v1-small`) ottiene risultati migliori su alcune dimensioni specifiche, come le *precisioni* sui diversi livelli di n-gram, la penalità di brevità, il length ratio e alcune misure legate all’allineamento tra

testo generato e riferimento.

Tali risultati possono essere spiegati da diversi fattori:

- **Bilanciamento tra capacità modellistiche e sintesi delle risposte:** TEXT_GEN_77M risulta essere abbastanza capiente da cogliere pattern articolati di risposta, ma non così grande da generare sistematicamente output ridondanti. L’architettura e il pretraining influenzano la produzione di risposte più concise, spesso preferite dai calcoli di precisione e da penalità di brevità elevate.
- **Output particolarmente concisi e focalizzati:** il modello tende a produrre risposte relativamente brevi e centrate, come evidenziato dal valore particolarmente basso di brevity penalty e length ratio. Questo porta a un incremento delle precisioni n-gram, specialmente quando le risposte attese sono sintetiche o molto specifiche — tipico nei contesti biomedici.
- **Sinergia tra modulo RAG e modello generativo:** un modello con meno parametri rispetto a grandi architetture, ma comunque superiore ai più piccoli, si dimostra più efficiente nell’incorporare informazioni dai chunk recuperati, sfruttando meglio il contesto fornito dal retriever senza venir “distratto” o saturato da dettagli inutili.
- **Specificità del fine-tuning o dataset:** in presenza di risposte gold standard particolarmente concise, oppure in presenza di valutatori automatici che penalizzano eccessivamente generazioni verbose, modelli come TEXT_GEN_77M possono risultare favoriti. Anche il mix tra composizione dei dati e impostazione della loss durante il fine-tuning può avvantaggiare modelli che puntano alla sintesi estrema.
- **Effetto dell’allineamento lessicale:** metriche come precision o brevity penalty premiano output che corrispondono esattamente, in modo compatto, alle espressioni attese, qualità che TEXT_GEN_77M sembra aver ottenuto più frequentemente in questa sperimentazione, a differenza di altri modelli più “dispersivi”.

5.1.4 Valutazione umana delle risposte e analisi qualitativa dei chunk recuperati

Oltre alle metriche automatiche, le performance dei modelli sono state valutate anche tramite ispezione e confronto manuale. Un essere umano ha verificato le risposte generate dai tre modelli rispetto alle risposte attese del dataset di test, analizzando sia la coerenza che la correttezza semantica delle risposte prodotte.

Dall’analisi qualitativa è emerso che tutti i modelli, pur presentando leggere differenze stilistiche e di formulazione, sono stati generalmente in grado di fornire risposte contestualmente corrette alla domanda posta. Questo suggerisce che la variabilità tra modelli riguarda prevalentemente la forma linguistica piuttosto che la sostanza informativa fornita.

Durante il processo di retrieval, è stata inoltre valutata la qualità dei chunk di testo selezionati dal sistema: circa la metà dei chunk recuperati (nell'ordine del 50%) corrispondevano effettivamente ai riferimenti attesi in riferimento alla risposta gold-standard del dataset. Questo dato testimonia sia la buona capacità del sistema di individuare informazioni rilevanti, sia la presenza di margini di miglioramento nella fase di retrieval, soprattutto nella precisione della selezione dei contesti informativi.

5.1.5 Conclusione

In definitiva, l'esperienza dimostra che la scelta di un modello meno capiente, ma meglio allineato al dominio, può risultare estremamente vantaggiosa su task specifici. Questo sottolinea l'importanza di preferire modelli compatti, ben addestrati e adattati, rispetto a soluzioni più grandi e generiche, specie in presenza di risorse computazionali limitate e dataset settoriali.

5.2 Limiti computazionali e problematiche riscontrate

Lo sviluppo e la sperimentazione del sistema RAG sono stati caratterizzati da diverse limitazioni legate alle risorse hardware e all'effettiva gestibilità dei modelli. Le principali difficoltà affrontate si riferiscono a problemi di memoria, prestazioni computazionali e scalabilità sia nella fase di fine-tuning che in quella di inference, con particolare evidenza quando si sono utilizzati modelli fino a 300M di parametri.

- **Risorse di memoria limitate:** La disponibilità di macchine con 8GB e 12GB di RAM si è rivelata spesso insufficiente per la gestione agevole di modelli di dimensioni medio-grandi, sia durante il fine-tuning sia nell'inferenza batch. L'occupazione di memoria aumentava rapidamente durante l'elaborazione di batch di dati o nelle fasi di caricamento degli embedding e dell'indice vettoriale FAISS.
- **Problemi di computazione e lentezza dei processi:** Anche nei casi in cui la memoria non era colma, la velocità di calcolo rappresentava un collo di bottiglia evidente. Pur avendo a disposizione un processore Apple M2 Pro, si è osservato che tale architettura (pur più prestante della M3 in nostro possesso) riscontrava comunque difficoltà nel gestire efficientemente modelli molto grandi. Il tempo necessario per epoche di fine-tuning complete, e anche per la procedura di retrieval e generazione sul test set, risultava spesso elevato.
- **Scelta obbligata del modello:** Proprio a causa di queste limitazioni, è stato possibile testare modelli con una taglia massima di circa 300 milioni di parametri (BYT5_SMALL_300M) mentre tentativi con modelli ancora più grandi sono risultati impraticabili per saturazione della memoria e pattern di swap eccessivamente penalizzanti.
- **Limiti anche sulla componente RAG:** Non solo il fine-tuning dei modelli generativi, ma anche la componente RAG ha sofferto delle limitate risorse: la fase di retrieval risulta rapida su porzioni di corpus limitate, ma subisce rallentamenti e

rischio di out-of-memory all'aumentare del volume dei dati indicizzati o delle query batchate.

In sintesi, la configurazione hardware disponibile (macchine consumer con RAM limitata e, seppur performanti, CPU/GPU non di classe server) ha condizionato l'intero workflow sperimentale, imponendo scelte conservative sulle dimensioni dei modelli utilizzati e costringendo a ottimizzare batch size, caching e modalità di processamento per evitare errori critici di memoria o drastiche penalizzazioni sui tempi di esecuzione.

6 Sviluppi Futuri

6.1 Potenziali sviluppi futuri

Alla luce dei risultati ottenuti e delle problematiche incontrate, questa sezione esplora alcune direzioni per possibili sviluppi futuri del sistema di domanda-risposta generativo (**RAG**), allo scopo di migliorarne le prestazioni, la scalabilità e l'adattamento a nuovi domini. Questi sviluppi sono stati identificati sia sulla base delle analisi condotte, sia rifacendosi a spunti emergenti dal progetto descritto nel documento “*NLP Generative Task QnA*”.

6.1.1 Utilizzo di modelli e strategie avanzate di retrieval

Una delle principali opportunità di miglioramento riguarda il modulo di **retrieval**, che attualmente si basa su un sistema FAISS. Per ridurre il margine di errore nella selezione dei chunk rilevanti e aumentare la precisione complessiva, sarebbe utile:

- Sperimentare sistemi di retrieval basati su modelli embedding più avanzati, come quelli ottimizzati per *Dense Passage Retrieval*, che includono un migliore supporto per contesti altamente strutturati.
- Integrare approcci ibridi che combinano il retrieval semantico con metodi pattern-matching tradizionali, migliorando recall e precision per domande complesse.
- Esplorare database vettoriali più scalabili come Weaviate⁸ o Pinecone⁹, che potrebbero migliorare la velocità e la gestione delle risorse per dataset più ampi.

Ridimensionamento dei modelli generativi

Le limitazioni delle risorse hardware disponibili nel nostro setup hanno imposto l'uso di modelli con un massimo di 300 milioni di parametri. Per risolvere questo problema, possiamo implementare le seguenti strategie:

- Adottare tecniche di quantizzazione avanzata e pruning per ridurre le richieste di memoria senza sacrificare le prestazioni.
- Addestrare modelli generativi su nodi cloud con GPU di classe server, consentendo di utilizzare modelli di dimensioni maggiori, come T5-large o GPT-like architectures per generare output di maggiore qualità.

⁸<https://python.langchain.com/docs/integrations/vectorstores/weaviate/>

⁹<https://python.langchain.com/docs/integrations/vectorstores/pinecone/>

6.1.2 Espansione delle metriche di valutazione

Attualmente, le performance del sistema sono valutate attraverso metriche automatiche (ROUGE, BLEU, METEOR) e revisioni umane delle risposte e dei chunk recuperati.

Tuttavia, è possibile espandere il set di metriche con:

- Metriche specifiche per il dominio biomedico, che valutano l'accuratezza e la correttezza tecnica del contenuto generato.
- Sistemi di valutazione end-to-end basati su framework come MTEB (Massive Text Embedding Benchmark) o score di ranking personalizzati.
- Metriche di explainability per esaminare come il modello utilizza i chunk recuperati durante la generazione delle risposte.

6.1.3 Adattamento a nuovi domini

Lo sviluppo futuro del sistema potrebbe prevedere l'estensione del modello ad altri domini specialistici oltre il biomedico. Questo potrebbe richiedere:

- Curazione di nuovi dataset settoriali e creazione di training pipeline specifiche per ogni dominio.
- Adattamento dei parametri del retrieval per contesti interdisciplinari con informazioni meno strutturate.
- Utilizzo di modelli pre-addestrati multi-dominio con capacità di generalizzazione estesa.

6.1.4 Ulteriori ottimizzazioni e automazione

Infine, un'importante area di sviluppo è rappresentata dall'automazione e dall'ottimizzazione del sistema:

- Automazione del processo di fine-tuning con setup che permettano l'addestramento continuo (*continual learning*) man mano che nuovi dati diventano disponibili.
- Miglioramento dell'orchestrazione delle pipeline per il parallelismo distribuito, aumentando l'efficienza delle operazioni RAG su larga scala.
- Implementazione di assistenti interattivi real-time con inferenza locale, ottimizzati per dispositivi edge.

Conclusione:

I futuri sviluppi del sistema RAG possono sfruttare una combinazione di tecniche avanzate di retrieval, modellazione generativa scalabile, e ottimizzazioni infrastrutturali per migliorare ulteriormente accuratezza, efficienza e versatilità su larga scala. Espandendo il focus verso approcci basati su automazione e riduzione dei costi computazionali, sarà possibile applicare il sistema a contesti più complessi con prestazioni sempre più avanzate e affidabili.