ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria e Scienze Informatiche

Integrazione di RAG e LLM nello Sviluppo del Software

Tesi di laurea in:
Programmazione ad oggetti

Relatore
Prof. Viroli Mirko

Candidato
Bollini Simone

Correlatori
Dott. Aguzzi Gianluca
Dott. Farabegoli Nicolas

IV Sessione di Laurea
Anno Accademico 2023-2024

Abstract

I Large Language Model (LLM) addestrati per sviluppare il codice sono oggi altamente efficaci e in grado di generare soluzioni utili e funzionanti. L'addestramento fatto sui modelli è però su fonti e soluzioni generali, questo non da quindi la possibilità al modello di generare soluzioni su misura per una specifica richiesta utilizzando casistiche già create dal programmatore o dalla propria azienda per casi simili. Da questo nasce l'esigenza di addestrare il modello per personalizzare le soluzioni proposte, contestualizzandole alla propria realtà aziendale e al proprio stile nel programmare. Il fine-tuning di un LLM è un processo molto costoso e non scalabile per essere aggiornato frequentemente, inoltre non è possibile addestrare il modello su tutte le casistiche possibili. Per rispondere a questa esigenza entra in gioco la Retrieval-Augmented Generation (RAG), che permette di recuperare informazioni da una base di conoscenza esterna al modello, come librerie specifiche di un azienda, arricchendo il contesto della query. L'output di questa **matrice di conoscenza** si inserisce e completa la query inviata al LLM, estendendo la base di informazioni sulla quale genererà l'output con la risposta. Questa tesi approfondisce questi concetti e sperimenta l'integrazione di un RAG con un LLM con lo scopo di ottenere dal LLM risposte personalizzate che solo con la conoscenza del LLM anche se estremamente performante e preparato sarebbe stato impossibile ottenere.

A Giulia e ai miei figli, il dono più grande.
A tutta la mia famiglia.

Grazie a tutti voi.

Contents

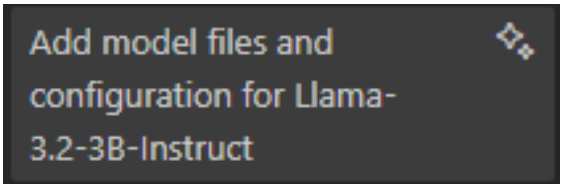
List of Figures

Chapter 1

Introduzione

1.1 Essere programmatori nel 2025

Sono disponibili tantissimi (IDE) per lo sviluppo del codice uno di questi è **Visual Studio Code**, mentre **Github** può essere lo strumento utilizzato condividere progetti per lavore in maniera collaborativa. Se richiesta memoria GPU per piccoli progetti accademici è disponibile **COLAB** che permette di eseguire in remoto codice offrendo anche gratuitamente utilizzo di GPU. Questi esempi mostrano una panoramica di strumenti vasta, complessa e in rapita evoluzione, con un frequente cambio di software per realizzare un programma. Un esempio d'utilizzo con gli strumenti sopra elencato potrebbe essere la realizzazione iniziale del progetto in locale utilizzado Visual Studio Code per poi riportare il tutto su GitHub. In un secondo momento il codice viene ripreso e aperto su Colab dove a sua volta il programma viene modificato ed infine rieseguito il Push sul progetto radice presente su GitHub. Ora nel 2025, la cosa che accomuna questi strumenti, è l'implemazione al loro interno di funzioni che basate sull'IA, in grado di completare il codice, suggerire correzioni e creare documentazione pertinente. Un esempio semplice ma che offre già un idea della vastità e della potenza di queste funzioni è l'utility di **Github Copilot** 'Generate Commit Message with Copilot' che propone il testo da utilizzare come descrizione di un commit, ho provato a riscontrare quanto fosse contestualizzato e coerente con quanto aggiornato e ho ottenuto il seguente risultato:



Nel mio caso quanto proposto era corretto ed ho quindi eseguito il Commit con la descrizione proposta. Quanto è riuscito a fare Copilot è strabiliante, in pochi istanti ha analizzato il contesto ritornando come output una risposta semplice ma coerente rispetto a quanto cambiato. L'uso di questi strumenti sta rendendo il lavoro molto più dinamico e veloce, riducendo le interruzioni nel cercare soluzioni o per trovare le giuste parole per descrivere quanto fatto.

L'intelligenza artificiale sta rivoluzionando il modo in cui il software viene sviluppato, strumenti come Copilot utilizzando tutto il loro potenziale, possono creare la spina dorsale di un progetto in pochi secondi lasciando al programmatore il compito di verificare e correggere solo in parte il codice proposto. In progetti complessi questo non riduce il ruolo del programmatore, anzi lo eleva a compiti di precisione e ad alto valore aggiunto lasciando la stesura di parti del codice semplici e ripetitive al software stesso. Sapere cosa chiedere e formulare correttamente le domande al LLM è fondamentale, esplicitando nel dettaglio con parole chiave mirate come deve essere realizzato il codice. Altro compito complesso per il

programmatore è non farsi troppo ammaliare dalle soluzioni proposte perché non sempre necessarie per quanto richiesto oppure diverse da quanto già conosciuto per realizzare una determinata funzione. Questo nuovo modo di lavorare per mette di conoscere nuove soluzioni ma comporta test e tempo non sempre disponibile, il programmatore deve sempre avere il controllo del progetto accettando generazione del codice automatica solo dove consapevole di quanto proposto e del suo impatto anche in casi di revisione e manutenzione futuri. L'ultimo miglio da percorrere per sfruttare questi strumenti è la personalizzazione delle risposte del LLM, per ottenere risposte coerenti con quanto già realizzato e conosciuto, per fare questo entra in gioco la RAG.

Chapter 2

Addestrare un LLM per la Generazione del Codice

L'addestramento di LLM per la generazione di codice di programmazione richiede una serie di passaggi metodici e risorse computazionali significative. Conoscere questo processo è utile per la successiva integrazione con la RAG. La procedura si divide nelle selle seguenti fasi:

2.1 Raccolta e Preparazione dei Dati

La qualità e la quantità dei dati per l'addestramento è di primaria importanza per prepare un modello alla generazione di codice in maniera efficace. È quindi essenziale utilizzare per il training codice sorgente proveniente da molteplici fonti tra cui codice sorgente, file Readme, documentazione tecnica, commenti nel codice, pagine Wiki, API e discussioni su forum specializzati in programmazione. In rete è possibile trovare diverso materiale open source tra cui dataset già etichettati. Alcuni dataset hanno un valore altissimo, per tutelare il costo per produrli per certi dataset è previsto il diritto d'autore. I dati si dividono in due tipologie:

- **Dati Strutturati**: seguono un formato specifico e predefinito.
- **Dati non Strutturati**: non sono organizzati e sono quindi più difficili da interpretare dal modello.

La raccolta di dati va visionata con cura, se non si conosce la provenienza del codice è possibile che contenga bug o codice opsoleto che possono essere trasmessi al modello. I dati raccolti devono essere quindi puliti e pre-processati per rimuovere errori e informazioni non pertinenti, garantendo così un dataset di alta qualità per l'addestramento. Sui dataset viene utilizzato un tokenizer specializzato che riconosce costrutti di programmazione come keyword, operatori e strutture sintattiche.

2.2 Pre-Addestramento

Il pre-addestramento di un LLM da utilizzare per la generazione di codice richiede un approccio specifico. A differenza del pre-addestramento generico, utilizzando i dataset precedentemente prepareti il modello impara a:

- Predire il completamento del codice
- Comprendere la struttura sintattica dei linguaggi di programmazione
- Riconoscere pattern comuni nel codice
- Identificare le relazioni tra diversi blocchi di codice

Un esempio pratico di pre-addestramento può essere implementato utilizzando la libreria transformers [WDS+20, FGT+20]:

```
from transformers import RobertaConfig, RobertaTokenizerFast

# Configurazione del modello per il codice
config = RobertaConfig(
    vocab_size=50000,  # Dimensione del vocabolario
    max_position_embeddings=514,  # Lunghezza massima sequenza
    num_attention_heads=12,  # Teste di attenzione
    num_hidden_layers=6,  # Strati nascosti
    type_vocab_size=1  # Tipo di vocabolario
)

# Tokenizer specializzato per il codice
tokenizer = RobertaTokenizerFast.from_pretrained(
    "microsoft/codebert-base",
    max_length=512,
    truncation=True,
```

```
    padding=True
)
```

Durante questa fase, il modello sviluppa una comprensione profonda della sintassi e della semantica del codice, che verrà poi raffinata durante il fine-tuning per compiti specifici di generazione del codice.

2.3 Fine-Tuning

Il fine-tuning è la fase in cui il modello viene specializzato per la generazione di codice, documentazione e risposta a quesiti specifici del contesto di programmazione. Durante questa fase, il modello affina le sue capacità attraverso:

- **Dataset Specializzati**: Utilizzo di dataset contenenti:
  - Coppie di descrizioni-implementazioni
  - Documentazione tecnica e commenti
  - Esempi di bug fixing e refactoring
- **Tecniche di Apprendimento**:
  - **Apprendimento Supervisionato**: Training su coppie input-output predefinite
  - **Apprendimento per Rinforzo**: Ottimizzazione basata su feedback e metriche di qualità
  - **Few-shot Learning**: Adattamento a nuovi contesti con pochi esempi

Un esempio pratico di fine-tuning può essere implementato utilizzando la libreria transformers [WDS+20]:

```
from transformers import Trainer, TrainingArguments
from datasets import load_dataset

# Caricamento del dataset per il fine-tuning
dataset = load_dataset("code_search_net", "python")

# Configurazione del training
training_args = TrainingArguments(
```

```
    output_dir="./results",
    num_train_epochs=3,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    warmup_steps=500,
    weight_decay=0.01,
    logging_dir="./logs",
    logging_steps=10,
    evaluation_strategy="epoch"
)

# Inizializzazione del trainer
trainer = Trainer(
    model=model,                 # Modello pre-addestrato
    args=training_args,          # Argomenti di training
    train_dataset=dataset["train"],
    eval_dataset=dataset["validation"],
    tokenizer=tokenizer,         # Tokenizer specializzato per il codice
)

# Avvio del fine-tuning
trainer.train()
```

Durante il fine-tuning, il modello sviluppa capacità specifiche come:

- Generazione di codice a partire da descrizioni in linguaggio naturale
- Completamento intelligente del codice basato sul contesto
- Creazione di documentazione tecnica
- Identificazione e correzione di bug
- Refactoring del codice seguendo best practices

Il processo di fine-tuning richiede un attento bilanciamento tra:

- **Overfitting**: Evitare che il modello memorizzi i dati di training
- **Generalizzazione**: Mantenere la capacità di adattarsi a nuovi contesti
- **Prestazioni**: Ottimizzare la velocità e la qualità delle risposte

2.4 Pre-Addestramento vs Fine-Tuning

È importante comprendere la distinzione tra queste due fasi dell'addestramento:

2.4.1 Pre-Addestramento

Il pre-addestramento è la fase iniziale dove il modello:

- Acquisisce una comprensione **generale** del linguaggio di programmazione
- Viene addestrato su **grandi quantità** di codice sorgente generico
- Impara le strutture base e la sintassi del linguaggio
- Non è ancora specializzato per compiti specifici

2.4.2 Fine-Tuning

Il fine-tuning è invece la fase di specializzazione dove il modello:

- Si adatta a un **dominio specifico** o a compiti particolari
- Utilizza dataset più piccoli ma **mirati**
- Affina le conoscenze per generare codice per specifici casi d'uso

Analogia: Si può paragonare a:

- Pre-addestramento: Imparare la grammatica e il vocabolario di base di una lingua
- Fine-tuning: Specializzarsi nel linguaggio tecnico di un settore specifico

2.5 Architettura del Modello

Gli LLM utilizzano tipicamente architetture basate su trasformatori, che sono particolarmente efficaci nell'elaborazione di sequenze di dati, come il testo e il codice. I trasformatori utilizzano meccanismi di auto-attenzione per valutare l'importanza di diversi elementi in una sequenza, permettendo al modello di comprendere le relazioni tra parole o token. Questa capacità è fondamentale nella generazione del codice, poiché le dipendenze tra variabili e funzioni possono estendersi su ampie sezioni del codice, richiedendo al modello di considerare un ampio contesto per trovare le risposte corrette.

2.6 Valutazione e Ottimizzazione

Una volta addestrato, il modello deve essere rigorosamente valutato utilizzando metriche specifiche per la generazione di codice, come la correttezza sintattica, la funzionalità e l'efficienza del codice prodotto. I risultati della valutazione possono essere utilizzati per ulteriori ottimizzazioni, come aggiustamenti dei pesi del modello, modifiche all'architettura o includere dati di addestramento aggiuntivi per affrontare eventuali carenze.

2.6.1 Metriche di Valutazione

- **Correttezza Sintattica**: Verifica che il codice generato sia sintatticamente corretto.
- **Funzionalità**: Verifica che il codice generato realizzi la funzionalità desiderata.
- **Efficienza**: Valuta le prestazioni del codice in termini di tempo di esecuzione e utilizzo delle risorse.

2.6.2 Tecniche di Ottimizzazione

- **Aggiustamento dei Pesi**: Modifica dei pesi del modello per migliorare le prestazioni.
- **Modifiche all'Architettura**: Introduzione di nuove componenti o modifiche a quelle esistenti.
- **Integrazione di Dati Aggiuntivi**: Utilizzo di ulteriori dati di addestramento per migliorare le prestazioni.

Chapter 3

RAG

3.1 Introduzione

Il RAG **Retrieval-Augmented Generation**, (in italiano *Generazione Aumentata tramite Recupero*) è un sistema che permette di migliorare l'output di un LLM estendendo la sua conoscenza con nuove informazioni, al di fuori dai suoi dati di addestramento. Allo scopo di:

- ottenere risposte personalizzate provenienti da librerie e codice custom;
- migliorare il codice generato rendendolo più specifico al dominio riducendo le allucinazioni;
- facilitare l'assistenza da parte del modello nella fase di debugging migliorando la sua comprensione di sistemi complessi;
- supportare la creazione di documentazione aggiornata;
- permettere all'interno di un Team di migliorare la coerenza del codice scritto da diversi programmatori;
- realizzare naturalmente senza forzature, codice più moderno proponendo librerie e standard comuni.
- evitare risposte imprecise a causa della confusione terminologica, in cui diverse fonti utilizzano la stessa terminologia per parlare di cose diverse.

3.2 Funzionamento

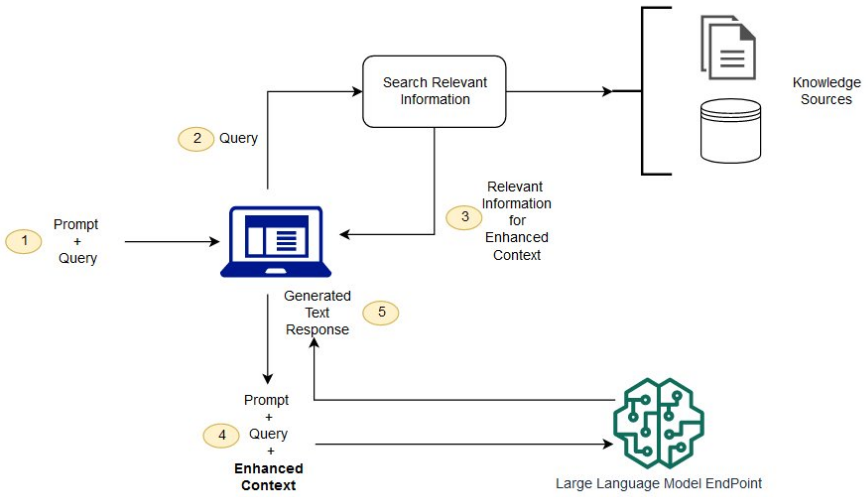


Figure 3.1: Flusso di una request ad un LLM integrato con un RAG

RAG è un sistema che integra il processo di generazione del linguaggio con un meccanismo di recupero delle informazioni. Il funzionamento si articola in diverse fasi, come illustrato in fig. 3.1.

3.2.1 Creazione degli Embedding

Il sistema RAG utilizza dati esterni al training set originale del LLM, provenienti da diverse fonti come:

- API e database interni
- Archivi documentali
- File di testo e codice

Questi dati vengono convertiti in rappresentazioni numeriche (embedding) e archiviati in un database vettoriale, creando una knowledge base accessibile dal RAG.

3.2.2 Fase 1: Function Calling

Il sistema RAG inizia con una chiamata di funzione per ricercare nei dati di embedding:

- La query dell'utente attiva una chiamata di funzione
- Il sistema cerca nei dati di embedding le informazioni pertinenti
- Se trovate, queste informazioni vengono aggiunte al prompt

3.2.3 Fase 2: Recupero delle Informazioni

Quando l'utente sottopone una query:

- La domanda viene convertita in un vettore
- Il sistema cerca nel database vettoriale le informazioni più pertinenti
- Viene calcolata la rilevanza attraverso calcoli matematici vettoriali

3.2.4 Fase 3: Aumento del Prompt

Il sistema RAG arricchisce il prompt dell'utente:

- Aggiunge le informazioni recuperate al contesto
- Utilizza tecniche di prompt engineering per ottimizzare la comunicazione con il LLM
- Fornisce al modello un contesto arricchito per generare risposte più accurate

3.3 Gestione dell'Aggiornamento dei Dati

Per mantenere l'efficacia del sistema nel tempo:

- Ricalcolo degli embedding per i nuovi dati
- Possibilità di aggiornamenti in tempo reale o batch

Questo approccio permette di superare le limitazioni dei LLM, fornendo risposte più accurate e contestualizzate grazie all'integrazione di conoscenze esterne aggiornate.

Chapter 4

Caso Studio: Implementazione di un Sistema RAG per lo Sviluppo del Software

4.1 Obiettivi del Caso Studio

Questo caso studio si propone di verificare il livello di personalizzazione e qualità delle risposte di un LLM potenziando la query nel prompt di input attraverso la creazione di un sistema RAG di supporto. Il sistema RAG è testato con della **classi JAVA uniche** create appositamente per il caso studio.
Problematica da affrontare: chiamate a più livelli di classi e metodi, dove il RAG potrebbe non essere in grado di estrapolare le informazione necessarie da inserire nel prompt per ottenere dal LLM risposte coerenti con quanto richiesto.

4.2 Architettura del Sistema

Il sistema RAG implementa un'architettura modulare composta da cinque componenti principali:

1. **Text Processor (Chunking)**:
    - Suddivide i file Java in chunk di un numero definito appositamente di

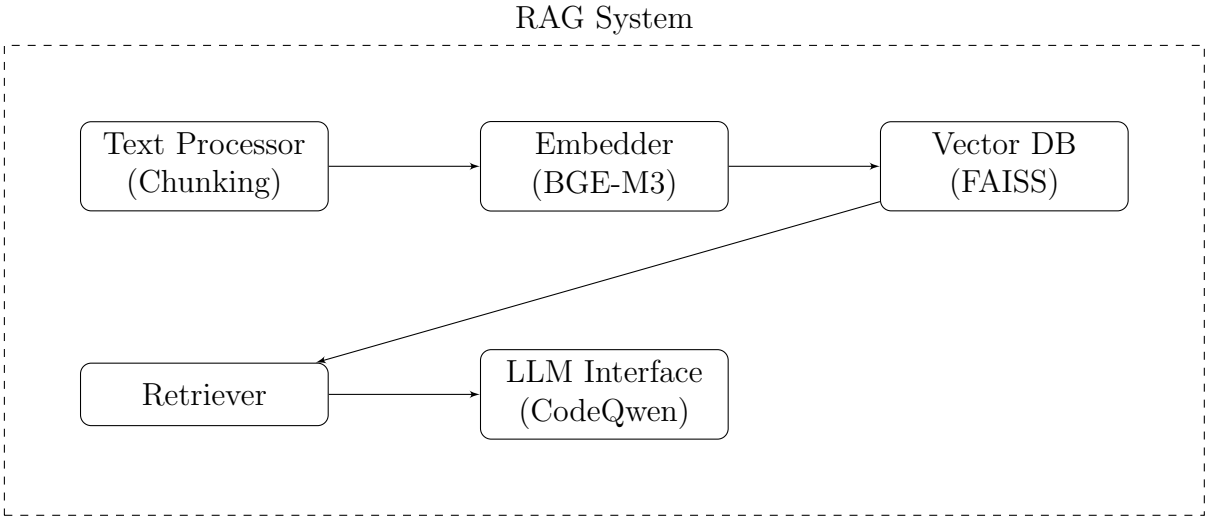


Figure 4.1: Architettura del sistema RAG

token

- Gestisce sovrapposizione di token tra chunk
- Preserva il contesto del codice

2. **Embedder (BGE-M3)**:
   - Converte i chunk in vettori numerici
   - Utilizza il modello BGE-M3 per la generazione degli embedding
   - Normalizza i vettori per ottimizzare la ricerca
3. **Vector DB (FAISS)**:
   - Memorizza gli embedding in un database vettoriale
   - Ottimizza la ricerca per similarità
   - Garantisce recupero efficiente dei chunk rilevanti
4. **Retriever**:
   - Esegue query semantiche sul database
   - Recupera i k chunk più rilevanti

- Prepara il contesto per il LLM

5. **LLM Interface (CodeQwen e Llama 3.2)**:
   - Interfaccia con i modelli CodeQwen e Llama 3.2
   - Genera risposte basate sul contesto recuperato
   - Ottimizza il prompt per la generazione di codice

4.3 Software Utilizzati

4.3.1 Ollama

Figure 4.2: Logo di Ollama

Ollama [Oll24] è un software che permette di utilizzare in locale LLM senza dover dipendere da servizi cloud esterni. Il software è stato scelto per la sua flessibilità, permettendo di integrare facilmente i modelli LLM nel sistema RAG.

4.3.2 LLM

Ogni LLM è specializzato per determinati scopi, per questo motivo per rendere più completa la ricerca sono stati utilizzati due modelli. Nella scelta dei modelli è stato obbligatorio eseguire un pre filtraggio considerando solo modelli che permettessero di eseguire function calling.

Llama 3.2

Llama 3.2 3B [AI24], un modello di linguaggio open source. Il modello, con 3 miliardi di parametri, è ottimizzato per compiti di dialogo multilingue e si distingue per le sue capacità di recupero e sintesi delle informazioni. La scelta è ricaduta su

questa versione per il suo equilibrio tra prestazioni e requisiti computazionali che permettono il suo utilizzo senza hardware troppo potente.

Codeqwen 1.5

Codeqwen [Tea24b] è un modello di linguaggio open source sviluppato da Qwen AI specializzato nella generazione di codice e documentazione tecnica. Con 7 miliardi di parametri, il modello è stato addestrato su un ampio dataset di codice sorgente e documentazione tecnica, permettendo di generare codice coerente e ben strutturato. La scelta di questo modello è stata dettata, a differenza di llama3.2, dalla sua specializzazione nella programmazione e dalla sua capacità di generare codice di alta qualità.

4.3.3 LangChain

LangChain [Tea24a] è un framework open source progettato per la gestione di dati strutturati e non strutturati in contesti multilingue. Fornisce strumenti avanzati per la creazione e la gestione di database vettoriali, facilitando l'organizzazione e la ricerca di informazioni in modo efficiente. LangChain è stato scelto per la sua capacità di integrare modelli di linguaggio con fonti di dati esterne.

4.3.4 BGE-M3

BGE-M3 [BAA24] è un database vettoriale open source per la gestione di dati strutturati e non strutturati multilingue, sviluppato da BigGraph Engine.

4.4 Dataset

Il dataset utilizzato per il caso studio è una classe custom di prova DateUtil.java simile ad una personalizzata utilizzata in qualsisi azienda per gestire il formato delle date.

4.5 Implementazione

4.5.1 Creazione dei Chunk

I modelli di embedding hanno limiti massimi di input (512-4096 token) per questo spezzare il codice in chunk di dimensioni adeguate è fondamentale. Inoltre occorre prestare attenzione alla dimensione dei chunk generati:

- Troppo piccoli: Riducono il contesto disponibile per il modello
- Troppo grandi: Chunk troppo grandi perdono focalizzazione semantica

Per suddividere il file Java in chunk viene utilizzata la libreria **langchain_text_splitters.** Il seguente codice Python mostra come suddividere il file Java in chunk di dimensione fissa, salvando i risultati in un file JSON.

Listing 4.1: Codice Python per la suddivisione del file Java in chunk

```python
from langchain_text_splitters import RecursiveCharacterTextSplitter
import json

# Carica il file Java
with open("my_project/DateUtil.java", "r") as f:
    java_code = f.read()

# Suddividi il codice in chunk
splitter = RecursiveCharacterTextSplitter(
    chunk_size=512,  # Dimensione di ogni chunk
    chunk_overlap=128,  # Overlap tra chunk
    separators=["\n\n", "\n", " ", ""]  # Separatori per il codice
)

chunks = splitter.split_text(java_code)

# Salva i chunk in un file JSON
chunks_data = [
    {
        "id": i + 1,
        "text": chunk,
        "source": "DateUtil.java",  # Nome del file
        "type": "code",             # Tipo di contenuto
        "start_line": 0,            # Linea di inizio (da calcolare)
        "end_line": 0               # Linea di fine (da calcolare)
    }
    for i, chunk in enumerate(chunks)
]
```

```
    with open("chunks.json", "w", encoding="utf-8") as f:
        json.dump(chunks_data, f, indent=4, ensure_ascii=False)
```

4.5.2 Generazione degli Embedding

Gli embedding trasformano i chunk in rappresentazioni vettoriali (array numerici) che catturano il significato semantico. Il seguente codice Python mostra come generare gli embedding e creare un database FAISS. FAISS (Facebook AI Similarity Search) è una libreria ottimizzata per la ricerca di similarità in spazi ad alta dimensionalità.

Listing 4.2: Codice Python per la generazione degli embedding e la creazione di un database FAISS

```
import json
from sentence_transformers import SentenceTransformer
from langchain_community.vectorstores import FAISS

# 1. Carica i chunk dal file JSON
with open("chunks.json", "r", encoding="utf-8") as f:
    chunks_data = json.load(f)

chunks = [item["text"] for item in chunks_data]

# 2. Carica il modello BGE-M3 e genera gli embedding
embedder = SentenceTransformer('BAAI/bge-m3')
embeddings = embedder.encode(chunks, show_progress_bar=True)

# 3. Crea un database FAISS
vector_store = FAISS.from_embeddings(
    text_embeddings=list(zip(chunks, embeddings)),  # Abbina testi e
        embedding
    embedding=embedder,
)

# 4. Salva il database
vector_store.save_local("./faiss_db")
print("Database FAISS creato e salvato in ./faiss_db.")
```

4.5.3 Esecuzione di una Query sul Database FAISS

Una volta creato il database FAISS, è possibile eseguire ricerche semantiche sui chunk memorizzati:

Listing 4.3: Codice Python per l'esecuzione di una query sul database FAISS

```
    from langchain_community.vectorstores import FAISS
    from langchain_community.embeddings import HuggingFaceEmbeddings

    # 1. Carica il modello di embedding nel formato corretto
    embedder = HuggingFaceEmbeddings(
        model_name="BAAI/bge-m3",
        model_kwargs={'device': 'cpu'},  # Usa 'cuda' per GPU
        encode_kwargs={'normalize_embeddings': True}
    )

    # 2. Carica il database FAISS esistente
    vector_store = FAISS.load_local(
        folder_path="./faiss_db",
        embeddings=embedder,
        allow_dangerous_deserialization=True
    )

    # 3. Query di esempio
    query = "Come formattare una data in Java?"
    docs = vector_store.similarity_search(query, k=3)

    for i, doc in enumerate(docs):
        print(f"Risultato {i+1}:")
        print(doc.page_content)
        print("-" * 40)
```

4.5.4 Creazione della Pipeline RAG

Il seguente codice Python mostra come configurare e utilizzare una pipeline RAG utilizzando FAISS, Ollama e i due LLM(LLAMA e CODEQWEN) per rispondere a domande specifiche sul codice Java:

Listing 4.4: Codice Python per la creazione della pipeline RAG

```
    from langchain_community.vectorstores import FAISS
    from langchain_community.embeddings import HuggingFaceEmbeddings
    from langchain_community.llms import Ollama
    from langchain.chains import RetrievalQA
    from langchain.prompts import PromptTemplate

# 1. Configurazione embedding
embedder = HuggingFaceEmbeddings(
    model_name="BAAI/bge-m3",
    model_kwargs={'device': 'cpu'},
    encode_kwargs={'normalize_embeddings': True}
)
```

```
13
14  # 2. Carica il database FAISS
15  vector_store = FAISS.load_local(
16      folder_path="./faiss_db",
17      embeddings=embedder,
18      allow_dangerous_deserialization=True
19  )
20
21  # 3. Configurazione modelli Ollama
22  LLAMA_TEMPLATE = """<|begin_of_text|>
23  <|start_header_id|>system<|end_header_id|>
24  Sei un esperto di programmazione. Rispondi in italiano basandoti esclusivamente
        sul contesto fornito.
25  Contesto: {context}<|eot_id|>
26  <|start_header_id|>user<|end_header_id|>
27  Domanda: {question}<|eot_id|>
28  <|start_header_id|>assistant<|end_header_id|>"""
29
30  CODEQWEN_TEMPLATE = """<|im_start|>system
31  Sei uno sviluppatore esperto. Fornisci risposte concise con codice basato sul
        contesto.<|im_end|>
32  <|im_start|>user
33  Contesto: {context}
34  Domanda: {question}<|im_end|>
35  <|im_start|>assistant
36  """
37
38  # 4. Funzione per selezionare il modello
39  def load_model(model_name):
40      models = {
41          "llama3": {
42              "template": LLAMA_TEMPLATE,
43              "params": {
44                  "temperature": 0.7,
45                  "system": "Rispondi in italiano come esperto di programmazione"
46              }
47          },
48          "codeqwen": {
49              "template": CODEQWEN_TEMPLATE,
50              "params": {
51                  "temperature": 0.3,
52                  "system": "Fornisci solo codice basato sul contesto"
53              }
54          }
55      }
56
57      if model_name not in models:
58          raise ValueError(f"Modello non supportato: {model_name}")
59
60      return Ollama(
```

```
        model=model_name,
        **models[model_name]["params"]
    ), PromptTemplate(
        template=models[model_name]["template"],
        input_variables=["context", "question"]
    )

# 5. Inizializza il modello
llm, prompt = load_model("codeqwen")

# 6. Catena RAG corretta
rag_chain = RetrievalQA.from_chain_type(
    llm=llm,
    chain_type="stuff",
    retriever=vector_store.as_retriever(
        search_kwargs={"k": 3, "score_threshold": 0.4}
    ),
    chain_type_kwargs={"prompt": prompt},
    return_source_documents=True,
    verbose=False
)

# 7. Funzione query
def ask_ollama(question):
    try:
        result = rag_chain.invoke({"query": question})

        print("\n\033[1;34mDOMANDA:\033[0m", question)
        print("\n\033[1;32mRISPOSTA:\033[0m")
        print(result["result"])

        print("\n\033[1;33mFONTI:\033[0m")
        for i, doc in enumerate(result["source_documents"], 1):
            print(f"{i}. {doc.page_content[:150]}...")
            if 'source' in doc.metadata:
                print(f"   Fonte: {doc.metadata['source']}")
            print("-" * 80)
    except Exception as e:
        print(f"\033[1;31mERRORE:\033[0m {str(e)}")

# 8. Esempio d'uso
if __name__ == "__main__":
    ask_ollama("Mostrami un esempio di formattazione della data in Java usando
        SimpleDateFormat")
```

4.5.5 AI Agent

E' un componente che può chiamare delle funzioni (function calling) nel mio caso search il LLM capisce se deve chiamare o meno il RAG.

Function Calling

Il "function calling" è un meccanismo che permette a un modello di linguaggio di determinare quando è necessario chiamare una funzione esterna e con quali parametri, in questo caso il RAG.

Chapter 5

Conclusioni

5.1 Risultati Ottenuti

da scrivere

5.2 Impatto sullo Sviluppo Software

L'integrazione di strumenti basati su AI nel processo di sviluppo software sta rivoluzionando il settore. Durante il periodo di sviluppo di questa tesi (Ottobre 2024 - Gennaio 2025), abbiamo osservato:

- Rapida evoluzione degli strumenti di AI per lo sviluppo software
- Crescente disponibilità di soluzioni open source
- Miglioramento continuo nelle capacità di generazione e comprensione del codice

5.3 Sfide e Prospettive Future

Bibliography

[AI24] Meta AI. Llama-3.2-3b: Open foundation and fine-tuned chat models, 2024.

[BAA24] BAAI. Bge-m3: A multi-modal model understanding images and text, 2024. HuggingFace model repository for BGE-M3, a multi-modal model for image and text understanding.

[Doc24] Huggingface Docs. Lora, dec 2024.

[Doc25] GitHub Docs. Asking github copilot questions in your ide, jan 2025.

[FGT+20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[Git24] GitHub. Github copilot is more than a tool, it's an ally, dec 2024.

[Met24] Meta. Llama-3.3-70b-instruct, dec 2024.

[Oll24] Ollama. Ollama documentation, 2024. GitHub repository.

[SBO23] Ahmed R. Sadik, Sebastian Brulin, and Markus Olhofer. Coding by design: Gpt-4 empowers agile model driven development, 2023.

[Tea24a] LangChain Team. Langchain documentation, 2024. Accessed: 2024-03-20.

[Tea24b] Qwen Team. Codeqwen1.5: A code-specialized language model, 2024. Accessed: 2024-03-20.

[Tea24c] Qwen Team. Qwen2.5-coder-3b: A code-specialized language model, 2024.

[WDS+20] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45. Association for Computational Linguistics, 2020.