Tutorial su RAG con LangChain e LangGraph | Shared Grok Conversation

Exported on 2025-12-29 01:02:55

Tutorial: Ricerca di Risposte da PDF o File di Testo con LangChain e LangGraph (RAG)

Ciao! Ecco un **bel tutorial passo-passo in italiano** su come creare un sistema di **Retrieval-Augmented Generation (RAG)** usando **LangChain** e **LangGraph**. Questo ti permetterà di caricare PDF o file di testo, indicizzarli e fare domande per ottenere risposte basate sul contenuto dei tuoi documenti.

Il focus è su un RAG semplice ma potente: carichiamo un PDF, lo dividiamo in chunk, creiamo embeddings, usiamo un vector store per la retrieval e integriamo tutto in un agente con LangGraph per un flusso più "agentico" (cioè, l'LLM decide se recuperare informazioni o rispondere direttamente).

Prerequisiti

- Python 3.10+
- Installa le librerie necessarie:
  text

```
pip install langchain langchain-community langchain-openai lang
```

- Una chiave API OpenAI (per embeddings e LLM). Impostala come variabile d'ambiente: export OPENAI_API_KEY=sk-...

Passo 1: Caricare il Documento (PDF o Testo)

Usiamo PyPDFLoader per PDF o TextLoader per file .txt.

Python

```
from langchain_community.document_loaders import PyPDFLoader  # O Te

# Esempio con PDF
loader = PyPDFLoader("percorso/al/tuo/documento.pdf")  # Sostituisci
documents = loader.load()

# Per file di testo: from langchain_community.document_loaders impor
# loader = TextLoader("percorso/al/tuo/file.txt", encoding="utf-8")

print(f"Caricati {len(documents)} pagine/chunk.")
```

Ogni document ha page_content (il testo) e metadata (es. numero pagina).

Passo 2: Dividere il Testo in Chunk

I documenti lunghi vanno divisi in pezzi più piccoli per embeddings efficienti.

Python

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,     # Dimensione chunk
    chunk_overlap=200    # Sovrapposizione per contesto
)

chunks = text_splitter.split_documents(documents)
print(f"Creati {len(chunks)} chunk.")
```

Passo 3: Creare Embeddings e Vector Store

Usiamo OpenAI embeddings e Chroma come vector database (locale e semplice).

Python

```
from langchain_openai import OpenAIEmbeddings
from langchain_chroma import Chroma

embeddings = OpenAIEmbeddings()  # Usa text-embedding-ada-002 di def
```

```
vectorstore = Chroma.from_documents(
    documents=chunks,
    embedding=embeddings,
    persist_directory="./chroma_db"  # Salva su disco per riutilizzo
)

retriever = vectorstore.as_retriever(search_kwargs={"k": 4})  # Recu
```

Passo 4: Creare una Catena RAG Semplice con LangChain

Prima un RAG base senza LangGraph.

Python

```
from langchain_openai import ChatOpenAI
from langchain import hub
from langchain_core.runnables import RunnablePassthrough
from langchain_core.output_parsers import StrOutputParser

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)

# Prompt standard per RAG (da LangChain hub)
prompt = hub.pull("rlm/rag-prompt")

def format_docs(docs):
    return "\n\n".join(doc.page_content for doc in docs)

rag_chain = (
    {"context": retriever | format_docs, "question": RunnablePassthr
    | prompt
    | llm
    | StrOutputParser()
)

# Test
risposta = rag_chain.invoke("Qual è il tema principale del documento
print(risposta)
```

Passo 5: Rendere "Agentico" con LangGraph

LangGraph permette di creare un agente che decide se recuperare dal vectorstore o rispondere direttamente (Agentic RAG).

Python

```python
from langgraph.graph import StateGraph, END
from typing import TypedDict, Annotated
import operator
from langchain_core.messages import HumanMessage

# Stato del grafo
class AgentState(TypedDict):
    messages: Annotated[list, operator.additem]

# Nodi
def retrieve(state):
    question = state["messages"][-1].content
    docs = retriever.invoke(question)
    return {"messages": [f"Contesto recuperato: {format_docs(docs)}"

def generate(state):
    llm_with_tools = llm.bind_tools([retriever])  # Se vuoi tool cal
    response = llm.invoke(state["messages"])
    return {"messages": [response]}

# Grafo
graph = StateGraph(AgentState)

graph.add_node("retrieve", retrieve)
graph.add_node("generate", generate)

# Logica condizionale: decidi se retrieve o generate
def decide(state):
    question = state["messages"][-1].content
    # Semplice regola: se contiene parole chiave, retrieve
    if any(word in question.lower() for word in ["documento", "pdf",
        return "retrieve"
    return "generate"

graph.add_conditional_edges("generate", decide, {"retrieve": "retrie
graph.add_edge("retrieve", "generate")
graph.set_entry_point("generate")

app = graph.compile()

# Test
inputs = {"messages": [HumanMessage(content="Riassumi il PDF")]}
```

```
for output in app.stream(inputs):
    print(output)
```

Questo agente è più flessibile: può recuperare solo quando serve!

Consigli Finali

- Per PDF complessi (con tabelle/immagini): prova UnstructuredPDFLoader o multimodal con GPT-4o.
- Per produzione: usa vectorstore persistenti come Pinecone o FAISS.
- Debug: usa LangSmith per tracciare le esecuzioni.
- Documentazione ufficiale: https://python.langchain.com/docs/tutorials/pdf_qa/

Prova con un tuo PDF e fammi sapere se hai domande! 😊

∅

È Meglio Usare Chroma? Dipende! Ecco una Spiegazione Dettagliata sui Vector Store per LangChain (2025)

Ciao! Chroma è un'ottima scelta in molti casi, ma non è sempre "la migliore" in assoluto. Dipende dal tuo progetto: se stai prototipando velocemente, lavorando localmente o con dataset piccoli/medi, **sì, Chroma è spesso la scelta ideale**. È semplice, open-source e si integra perfettamente con LangChain.

Ma per applicazioni in produzione, scalabili o con requisiti avanzati (es. filtraggio complesso, alta concorrenza, miliardi di vettori), ci sono alternative superiori.

LangChain supporta **decine di vector store** (vedi la lista ufficiale qui: https://python.langchain.com/docs/integrations/vectorstores/). L'interfaccia è unificata, quindi passare da uno all'altro è facile (basta cambiare una riga di codice).

Tabella di Confronto: Principali Vector Store per LangChain (Aggiornato al 2025)

Vector Store	Tipo	Pro	Contro	Quando Usarlo (Ideale per RAG con PDF/Testi)	Integrazione LangChain	Costo
Chroma	Locale/Open-source	- Super semplice e veloce da setup - Persistenza su disco - Ottima per prototyping - Integrazione nativa/deep con LangChain	- Non scalabile per dataset enormi (> milioni di vettori) - Problemi occasionali con filtraggio metadata - Non "production-ready" per alta concorrenza	Prototipi, app locali, sviluppo rapido, dataset < 1-5 milioni vettori	Eccellente (prima classe)	Gratuito
FAISS	Libreria locale (Facebook)	- Velocissimo per ricerca (soprattutto exact/flat) - Gestisce milioni di vettori su laptop - Nessun server needed	- Filtraggio metadata limitato (no nativo in LangChain base) - Non persistente di default (ma salvabile) - Meno features avanzate	App locali veloci, senza bisogno di filtering complesso, prototyping avanzato	Ottima	Gratuito
Pinecone	Cloud managed	- Scalabile a miliardi di vettori - Serverless (no gestione infra) - Alta affidabilità e performance - Metadata filtering eccellente	- Costoso (pod-based pricing) - Vendor lock-in - Dipende da internet	Produzione enterprise, alta scala, team senza ops	Ottima	A pagamento (da ~70$/mese)
Weaviate	Open-source (self-hosted o cloud)	- Hybrid search (vector + keyword) - Filtraggio	- Setup più complesso (Docker/K8s) - Curva di apprendimento	App con filtering complesso, hybrid search, self-	Buona	Gratuito (open) / Cloud paid

Vector Store	Tipo	Pro	Contro	Quando Usarlo (Ideale per RAG con PDF/Testi)	Integrazione LangChain	Costo
		avanzato e GraphQL - Moduli per embedding automatici - Buona per knowledge graph		hosted production		
Qdrant	Open-source (self-hosted o cloud)	- Alta performance e low latency - Filtering potente - Quantization per efficienza - Usato da big come OpenAI/X	- Richiede gestione server (se self-hosted)	Produzione scalabile, filtering avanzato, hybrid search	Ottima	Gratuito (open) / Cloud paid
pgvector	Estensione Postgres	- Integra con DB relazionale esistente - Filtraggio SQL potente - Production-ready e cheap	- Più lento su scala massive rispetto a dedicated vector DB - Setup DB required	Se usi già Postgres, app con dati strutturati + vettori	Buona	Gratuito (con Postgres)
Milvus	Open-source scalabile	- Gestisce miliardi di vettori - Distributed e high-throughput	- Complesso da deployare su scala	Enterprise massive scale, self-hosted	Buona	Gratuito (open)

Altri popolari: **MongoDB Atlas Vector Search**, **Redis**, **LanceDB** (buono per file-based), **Astra DB**.

Quando Usare Locale vs Cloud?

- **Locale (Chroma, FAISS, pgvector, Qdrant/Weaviate in Docker)**:
  - Prototipi, sviluppo, test.
  - Dati sensibili/privacy (tutto offline).
  - Costo zero.
  - Dataset piccoli/medi.
  - Esempio: Il tuo tutorial con PDF personali → Chroma o FAISS bastano!
- **Cloud (Pinecone, Weaviate Cloud, Qdrant Cloud, Pinecone)**:
  - Produzione con traffico alto.
  - Scalabilità automatica.
  - Team multipli, alta disponibilità.
  - Ma: Costi, dipendenza internet, possibili privacy issues.

Molti iniziano con Chroma/FAISS locale, poi migrano (es. esporta embeddings e importa in Pinecone/Qdrant).

Consiglio Personale per il Tuo Caso (RAG su PDF/Testi)

- **Inizia con Chroma**: È quello che ho usato nel tutorial precedente perché è il più semplice per caricare PDF, chunkare, embeddare e queryare. Perfetto per imparare e testare.
- **Se cresci**:
  - Vuoi velocità locale → Prova **FAISS** (cambia Chroma.from_documents in FAISS.from_documents).
  - Filtraggio per metadata (es. per pagina/autore) → **Qdrant** o **Weaviate**.
  - Produzione cloud senza pensieri → **Pinecone**.

Esempio rapido switch da Chroma a FAISS:

Python

```
# Invece di Chroma
# from langchain_chroma import Chroma
# vectorstore = Chroma.from_documents(...)

from langchain_community.vectorstores import FAISS
vectorstore = FAISS.from_documents(chunks, embeddings)
vectorstore.save_local("faiss_index")  # Persistenza
```

Se hai dettagli sul tuo progetto (es. dimensione dataset, produzione?, filtering needed?), ti do un consiglio più preciso! 😊

Spiegazione Dettagliata su FAISS e Adattamenti al Tuo Sistema RAG con LangChain

Ciao! Ti rispondo passo per passo alla tua query, in modo dettagliato e teorico dove serve. Inizierò spiegando FAISS, poi passerò agli adattamenti per più file (PDF e TXT), come gestire dati già nel DB, e infine un tutorial teorico su come funziona internamente la ricerca di risposte in un PDF (con focus su embeddings, vettori, retrieval). Userò esempi pratici, inclusa l'idea di esportare questa chat come PDF e queryare "tutorial langgraph pdf". Andiamo con ordine!

1. Parlami di FAISS

FAISS (Facebook AI Similarity Search) è una libreria open-source sviluppata da Meta (ex Facebook AI Research) per la ricerca efficiente di similarità su grandi insiemi di vettori ad alta dimensionalità. È progettata per gestire miliardi di vettori in modo veloce, usando algoritmi di indexing approssimativi (come IVFFlat, HNSW, PQ) che sacrificano un po' di precisione per guadagnare in velocità e scalabilità.

Perché è utile in un contesto come LangChain/RAG?

- In un sistema RAG (Retrieval-Augmented Generation), dopo aver creato embeddings (rappresentazioni numeriche del testo), hai bisogno di un "vector store" per memorizzarli e cercarli velocemente. FAISS è eccellente per questo: è locale (non richiede server cloud), velocissimo su hardware comune (CPU/GPU), e gestisce dataset medi/grandi senza problemi.
- **Pro**:
  - Alta performance: Ricerca in millisecondi su milioni di vettori.
  - Leggero: Funziona su laptop, senza dipendenze esterne pesanti.
  - Flessibile: Supporta vari indici (es. Flat per esattezza, IVFPQ per compressione).
  - Gratuito e open-source (licenza MIT).
  - Integrazione perfetta con LangChain: Basta una riga per usarlo.
- **Contro**:

- Non ha filtraggio metadata nativo avanzato (es. per autore/pagina), ma puoi aggiungerlo manualmente.
- Non persistente di default (ma salvi l'indice su disco).
- Meno scalabile di cloud come Pinecone per miliardi di vettori o concorrenza alta.
- Curva di apprendimento per configurare indici avanzati (ma in LangChain è semplificato).

Come si usa in pratica con LangChain?
Nel tuo codice precedente, invece di Chroma, fai così:

Python

```
from langchain_community.vectorstores import FAISS
from langchain_openai import OpenAIEmbeddings

embeddings = OpenAIEmbeddings()

# Supponi di avere i tuoi chunks pronti
vectorstore = FAISS.from_documents(chunks, embeddings)

# Salva su disco per riutilizzo
vectorstore.save_local("faiss_index")

# Per caricare un indice esistente
vectorstore = FAISS.load_local("faiss_index", embeddings, allow_dang

# Retriever (per ricerca)
retriever = vectorstore.as_retriever(search_kwargs={"k": 4})  # k =
```

- Configurazione avanzata: Puoi specificare l'indice, es. FAISS.from_documents(..., index_name="IVFFlat") per un indice inverted file.
- Dal 2025, FAISS ha aggiornamenti per GPU migliori e integrazione con PyTorch, ma in LangChain resta semplice.

Se il tuo progetto è locale e veloce, FAISS è meglio di Chroma per performance pura. Per info ufficiali, il repo GitHub è il posto migliore (ma non ho bisogno di tool per dirtelo, è conoscenza base).

2. Se Avessi 3 PDF e 2 TXT? Come Cambia il Loader?

Il loader singolo (PyPDFLoader) è per un file alla volta, ma per più file devi iterare o usare un loader multi-file. LangChain supporta caricamenti batch per efficienza.

Adattamento del Codice:

- Per PDF: Usa PyPDFLoader in un loop.
- Per TXT: Usa TextLoader.
- Poi, combina tutti i documenti in una lista unica, splittali in chunks, e crea il vectorstore.

Esempio completo:

Python

```
from langchain_community.document_loaders import PyPDFLoader, TextLo
from langchain_text_splitters import RecursiveCharacterTextSplitter

# Lista file (esempio: 3 PDF e 2 TXT)
pdf_files = ["pdf1.pdf", "pdf2.pdf", "pdf3.pdf"]
txt_files = ["txt1.txt", "txt2.txt"]

# Carica PDF
pdf_docs = []
for pdf in pdf_files:
    loader = PyPDFLoader(pdf)
    pdf_docs.extend(loader.load())  # Aggiungi a lista comune

# Carica TXT
txt_docs = []
for txt in txt_files:
    loader = TextLoader(txt, encoding="utf-8")
    txt_docs.extend(loader.load())

# Combina tutti
all_documents = pdf_docs + txt_docs

# Split in chunks (stesso come prima)
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chun
chunks = text_splitter.split_documents(all_documents)

# Poi procedi con vectorstore (es. FAISS o Chroma)
from langchain_community.vectorstores import FAISS
from langchain_openai import OpenAIEmbeddings
```

```
embeddings = OpenAIEmbeddings()
vectorstore = FAISS.from_documents(chunks, embeddings)
```

- **Perché così?** Ogni loader estrae testo e metadata (es. fonte file, pagina). Combinandoli, il retriever saprà da quale file viene un chunk (tramite metadata).
- **Bonus:** Per directory intere, usa DirectoryLoader:
  Python

  ```
  from langchain_community.document_loaders import DirectoryLoade

  loader = DirectoryLoader("cartella/", glob="**/*.pdf", loader_cl
  documents = loader.load()
  ```

  Aggiungi un altro per TXT.

Questo scala bene: per 3 PDF + 2 TXT, è efficiente. Se i file sono grandi, carica lazy (con loader.load_and_split()).

3. E Se Avessi Dati Già nel DB?

Se hai già un vectorstore (es. FAISS o Chroma salvato), non ricarichi da zero: lo carichi direttamente. Questo evita di ricalcolare embeddings (costoso in tempo/API).

Esempio con FAISS:

Python

```
# Carica esistente
vectorstore = FAISS.load_local("faiss_index", embeddings, allow_dang

# O con Chroma
from langchain_chroma import Chroma
vectorstore = Chroma(persist_directory="./chroma_db", embedding_func

# Poi usa il retriever come sempre
retriever = vectorstore.as_retriever()
```

- **Aggiungere nuovi dati:** Se vuoi aggiornare (es. aggiungi un nuovo PDF):
  Python

```
new_docs = PyPDFLoader("nuovo.pdf").load()
new_chunks = text_splitter.split_documents(new_docs)
vectorstore.add_documents(new_chunks)  # Aggiunge senza sovrasc
vectorstore.save_local("faiss_index")  # Salva update
```

- Per DB cloud (es. Pinecone), usa l'API per upsert (aggiorna/inserisci).

Questo rende il sistema persistente: carica una volta, querya infinite volte.

4. Tutorial Teorico: Cosa Succede Internamente nella Ricerca di Risposte (Focus su PDF, Embeddings, Vettori)

Ora entriamo nel teorico! Spiegherò come un sistema RAG trova risposte in un PDF (o TXT). È "magia" basata su matematica: testi → numeri → similarità.

Passo 1: Caricamento e Chunking (Preparazione)

- Carichi il PDF: Estrae testo grezzo (con PyPDFLoader). Un PDF è solo pagine di testo/immagini, ma noi prendiamo il testo.
- Dividi in chunks: Perché? Modelli come OpenAI hanno limiti (es. 4096 token). Chunk di 1000 chars + overlap (200) preservano contesto (es. frasi spezzate).

Passo 2: Embeddings – "Divisione di Parole in Numeri"

- **Cos'è un embedding?** Un vettore numerico (array di float, es. 1536 dimensioni per OpenAI ada-002) che rappresenta semanticamente il testo. Non è "parole in numeri" semplice (come ASCII), ma cattura significato.
  - Esempio: "Cane" potrebbe essere [0.1, -0.2, 0.3, ..., 0.5] (1536 valori).
  - Simili semanticamente sono vicini: "Cane" e "Cucciolo" hanno vettori con distanza bassa (cosine similarity alta).
- **Come si crea?** Usi un modello pre-addestrato (es. OpenAIEmbeddings): Invia chunk al API, torna vettore.
- Teoria: Basato su transformer (come BERT/GPT). Il modello "impara" da miliardi di testi a mappare parole/contesti in spazio vettoriale. Distanza euclidea o cosine misura similarità.

Passo 3: Indexing nel Vector Store (Memorizzazione)

- Salvi vettori + metadata (es. chunk testo, file origine) in FAISS/Chroma.

- FAISS crea un indice: Non cerca tutti i vettori (lento), ma usa strutture come alberi/k-means per approssimare.

Passo 4: Query e Retrieval (Trovare la Risposta)

- Utente fa domanda: es. "tutorial langgraph pdf".
- Embedda la query: Diventa un vettore Q.
- Cerca nel vectorstore: Trova i K chunk più simili (es. k=4) misurando distanza da Q.
  - Algoritmo: Cosine similarity = (Q · C) / (|Q| |C|), dove C è vettore chunk. Alto score = rilevante.
- Recupera chunk: Es. chunk con "LangGraph permette di creare un agente..." ha score alto perché semanticamente matcha.

Passo 5: Generation (Risposta Finale)

- Invia chunk recuperati + query all'LLM (es. GPT): "Rispondi basandoti su questo contesto: [chunks] alla domanda: [query]".
- LLM genera risposta grounded (non allucina, usa solo info recuperate).

Come Ti Accorgi di Aver Trovato la Risposta?

- Nel codice: Il retriever torna chunk con score (es. retriever.invoke(query)). Score > 0.8 (cosine) = buon match.
- In pratica: Stampa i chunk recuperati. Se contengono info rilevante, hai trovato; altrimenti, raffina (es. chunk più grandi, embeddings migliori).

5. Esempio Pratico: Esporta Questa Chat come PDF e Querya "Tutorial Langgraph PDF"

Supponi esporti questa conversazione (es. da browser a PDF, ~5 pagine).

Setup:

- Carica PDF: loader = PyPDFLoader("chat_export.pdf"); documents = loader.load() (estrae testo chat).
- Chunk: Splitta in pezzi (es. uno per messaggio).
- Embedda e indicizza con FAISS.

Query: "tutorial langgraph pdf"

- Embedda query → Vettore Q.
- Retrieval: Cerca chunk simili. Troverà chunk da miei messaggi precedenti (es. codice LangGraph con grafo, nodi retrieve/generate).

- Come trovi: Score alto su chunk contenenti "LangGraph", "PDF", "tutorial". Es. chunk: "Passo 5: Rendere 'Agentico' con LangGraph..." matcha semanticamente.
- Ti accorgi: Stampa docs = retriever.invoke("tutorial langgraph pdf"); vedi se docs hanno il codice/esempio. Se sì, LLM genera: "Nel tutorial, LangGraph crea un grafo con nodi per retrieve e generate...".
- Lavoro: Testa iterativamente. Se non trova (es. query vaga), usa hybrid search (vector + keyword) in store come Weaviate.

Questo è come "lavori": Prototipa, debugga retrieval (stampa chunk), adatta parametri. Se hai codice specifico da testare, dimmi! 😊

∅

Persistenza nei Vector Store Locali: Focus su FAISS e Chroma

Quando dico che **FAISS è "non persistente di default"**, intendo che **l'indice creato in memoria (RAM) sparisce quando chiudi il programma Python**. Se non fai nulla per salvarlo, ogni volta che riavvii lo script devi ricalcolare tutti gli embeddings e ricostruire l'indice da zero (molto costoso in tempo e soldi API).

La **persistenza** significa invece salvare l'indice (e gli embeddings associati) su disco, in modo da poterlo ricaricare velocemente in sessioni future senza rifare tutto.

Ecco come funziona in dettaglio per i due vector store locali più usati in LangChain: **FAISS** e **Chroma**.

1. FAISS – Persistenza Manuale

FAISS è una libreria pura per ricerca vettoriale: non ha un meccanismo automatico di salvataggio. Devi esplicitamente salvare e caricare l'indice.

Codice Pratico per Persistenza con FAISS

Python

```
from langchain_community.vectorstores import FAISS
from langchain_openai import OpenAIEmbeddings

embeddings = OpenAIEmbeddings()

# PRIMA VOLTA: Crea e salva
# (supponi di avere già i tuoi chunks pronti)
```

```
vectorstore = FAISS.from_documents(chunks, embeddings)

# SALVA su disco (crea una cartella con vari file binari)
vectorstore.save_local("cartella_faiss_index")    # <-- Qui rendi per

print("Indice salvato su disco!")
```

Python

```
 # VOLTE SUCCESSIVE: Carica senza ricalcolare embeddings
vectorstore = FAISS.load_local(
    folder_path="cartella_faiss_index",
    embeddings=embeddings,
    allow_dangerous_deserialization=True    # Necessario dal 2024+, p
)

retriever = vectorstore.as_retriever()
print("Indice caricato da disco in pochi secondi!")
```

Cosa viene salvato realmente?

- Una cartella (cartella_faiss_index) contenente:
  - index.faiss → la struttura dell'indice vettoriale (es. IVFFlat, HNSW).
  - index.pkl → un file pickle con i documenti originali, i metadata e gli ID.
- Dimensione: dipende dal numero di chunk. Con 1000 chunk da 1000 token ≈ 50-200 MB.

Vantaggi

- Caricamento velocissimo (secondi, anche con milioni di vettori).
- Puoi aggiornare: vectorstore.add_documents(nuovi_chunks) e poi save_local() di nuovo.

Svantaggi

- Devi gestire tu la cartella (backup, versione, ecc.).
- Se modifichi il modello di embedding (es. passi da ada-002 a un altro), l'indice vecchio diventa incompatibile.

2. Chroma – Persistenza Automatica (più comoda)

Chroma è costruito proprio per essere un vector database locale semplice e persistente. Per default **salva tutto automaticamente su disco**.

Codice Pratico per Persistenza con Chroma

Python

```
from langchain_chroma import Chroma
from langchain_openai import OpenAIEmbeddings

embeddings = OpenAIEmbeddings()

# PRIMA VOLTA: Crea con directory persistente
vectorstore = Chroma.from_documents(
    documents=chunks,
    embedding=embeddings,
    persist_directory="./chroma_db"    # <-- Cartella dove salva tutt
)

# Chroma salva automaticamente dopo ogni operazione!
print("Indice creato e già persistente nella cartella ./chroma_db")
```

Python

```
# VOLTE SUCCESSIVE: Carica direttamente
vectorstore = Chroma(
    persist_directory="./chroma_db",
    embedding_function=embeddings
)

retriever = vectorstore.as_retriever()
print("Indice caricato da disco automaticamente!")
```

Cosa viene salvato?

- Una cartella ./chroma_db con:
  - Un database SQLite interno (per metadata e documenti).
  - File binari per gli embeddings e l'indice HNNSW (molto efficiente).
- Più "pulito" e organizzato rispetto a FAISS.

Vantaggi di Chroma per persistenza

- Nessun comando extra: salva in automatico dopo ogni add_documents, from_documents, ecc.
- Puoi anche chiamare esplicitamente vectorstore.persist() (vecchio metodo, ora deprecato ma funziona).
- Facile da spostare/copiare la cartella intera.

Svantaggi

- Leggermente più lento di FAISS puro su dataset molto grandi.
- La cartella può crescere di più (SQLite overhead).

Confronto Diretto: Persistenza FAISS vs Chroma

Caratteristica	FAISS	Chroma
Persistenza di default	No (manuale)	Sì (automatica)
Comando per salvare	save_local("cartella")	Specifica persist_directory
Comando per caricare	load_local(...)	Solo istanziare con persist_directory
Facilità per principianti	Media	Alta
Performance su grandi dataset	Eccellente (più veloce)	Buona
Struttura file salvati	Cartella con .faiss + .pkl	Cartella con SQLite + binari HNNSW
Aggiornamenti frequenti	Buono (add + save)	Ottimo (automatico)

Consiglio Pratico per Te

- **Se stai imparando/prototipando velocemente** → Usa **Chroma** con persist_directory. È zero pensieri: crei una volta, poi ogni volta che avvii lo script lo ricarichi in 1-2 secondi.
- **Se vuoi massima velocità e controllo** → Usa **FAISS** con save_local/load_local. Ideale se hai tanti chunk e vuoi ottimizzare.

In entrambi i casi, una volta reso persistente:

- Puoi spegnere il computer.
- Puoi aggiungere nuovi PDF in futuro con add_documents().
- Non paghi più API OpenAI per embeddings già calcolati.

Se vuoi, ti scrivo uno script completo che:

1. Controlla se la cartella esiste.
2. Se sì → carica.
3. Se no → crea da zero e salva.

Fammi sapere quale preferisci (FAISS o Chroma)! 😊

∅

Tutorial Completo: Sistema RAG con LangChain e LangGraph Usando PDF/Testi e SQLiteVSS come Vector Store

Ciao! In base alle tue richieste precedenti (tutorial su RAG per ricerca risposte da PDF/file di testo con LangChain/LangGraph, caricamento multi-file, persistenza, e alternative a Chroma/FAISS), ti scrivo un **tutorial completo e pronto all'uso**. Poiché hai menzionato "su dbeaver ho sqlite", ho scelto **SQLiteVSS** come vector store: è basato su SQLite (persistente in un file .db), integra Faiss per ricerca vettoriale efficiente, e si integra perfettamente con LangChain. Puoi visualizzare e queryare il database direttamente in DBeaver (che supporta SQLite nativamente).

Questo tutorial copre:

- Caricamento multi-file (es. 3 PDF + 2 TXT).
- Divisione in chunk.
- Creazione embeddings.
- Indicizzazione persistente con SQLiteVSS (file SQLite .db).
- Catena RAG semplice.
- Agente agentico con LangGraph.
- Come visualizzare/analizzare il DB in DBeaver.

È in italiano, con codice Python testabile. Assumi di avere un ambiente Python pronto.

Prerequisiti

- Python 3.10+.
- Chiave API OpenAI (per embeddings e LLM): export OPENAI_API_KEY=sk-....
- Installa librerie:
  text

```
pip install langchain langchain-community langchain-openai lang
```

    - **Nota su sqlite-vss**: È un'estensione SQLite per vector search (usa Faiss internamente). Dopo install, carica automaticamente l'extension in SQLite.
- File esempio: Crea 3 PDF (es. "pdf1.pdf", "pdf2.pdf", "pdf3.pdf") e 2 TXT ("txt1.txt", "txt2.txt") in una cartella. O usa i tuoi!
- DBeaver: Assumi sia installato e configurato per SQLite (scarica driver se necessario).

Passo 1: Caricamento Multi-File (PDF e TXT)

Carichiamo più file e li combiniamo in una lista di documenti.

Python

```
from langchain_community.document_loaders import PyPDFLoader, TextLo

# Lista file esempio (adatta ai tuoi percorsi)
pdf_files = ["pdf1.pdf", "pdf2.pdf", "pdf3.pdf"]
txt_files = ["txt1.txt", "txt2.txt"]

# Carica PDF
pdf_docs = []
for pdf in pdf_files:
    loader = PyPDFLoader(pdf)
    pdf_docs.extend(loader.load())

# Carica TXT
txt_docs = []
for txt in txt_files:
    loader = TextLoader(txt, encoding="utf-8")
    txt_docs.extend(loader.load())

# Combina tutti
all_documents = pdf_docs + txt_docs

print(f"Caricati {len(all_documents)} documenti totali.")
```

Ogni documento ha page_content (testo) e metadata (es. fonte, pagina).

Passo 2: Divisione in Chunk

Dividiamo i documenti in pezzi piccoli per embeddings efficienti.

Python

```python
from langchain_text_splitters import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,     # Dimensione max chunk
    chunk_overlap=200    # Sovrapposizione per contesto
)

chunks = text_splitter.split_documents(all_documents)
print(f"Creati {len(chunks)} chunk.")
```

Passo 3: Creazione Embeddings

Usiamo OpenAI per embeddings (alternativa gratuita: SentenceTransformer come "all-MiniLM-L6-v2").

Python

```python
from langchain_openai import OpenAIEmbeddings

embeddings = OpenAIEmbeddings(model="text-embedding-ada-002")
```

Passo 4: Creazione Vector Store con SQLiteVSS (Persistente in SQLite)

SQLiteVSS crea un DB SQLite persistente (.db file). È locale, efficiente, e usa l'extension sqlite-vss per ricerca vettoriale.

Python

```python
from langchain_community.vectorstores import SQLiteVSS

# Crea il vector store da chunk (prima volta)
vectorstore = SQLiteVSS.from_documents(
    documents=chunks,
    embedding=embeddings,
    table="rag_table",          # Nome tabella nel DB
    db_file="./rag_vector.db"  # File SQLite persistente (crea se nc
)

# Per caricare un DB esistente (volte successive, senza ricalcolare
# vectorstore = SQLiteVSS(
#     table="rag_table",
```

```
#     embedding=embeddings,
#     db_file="./rag_vector.db"
# )

# Crea retriever per ricerca (recupera i 4 chunk più simili)
retriever = vectorstore.as_retriever(search_kwargs={"k": 4})

print("Vector store creato e persistente in ./rag_vector.db!")
```

- **Persistenza**: Il file ./rag_vector.db è un DB SQLite standard. Non sparisce al riavvio; carica velocemente.
- **Aggiungere nuovi dati**: vectorstore.add_documents(nuovi_chunks) aggiorna il DB automaticamente.
- **Internamente**: Crea una tabella con colonne come rowid, vector (BLOB binario per embeddings), content (testo chunk), e metadata.

Passo 5: Catena RAG Semplice

Una catena base per queryare e generare risposte.

Python

```
from langchain_openai import ChatOpenAI
from langchain import hub
from langchain_core.runnables import RunnablePassthrough
from langchain_core.output_parsers import StrOutputParser

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)

# Prompt RAG standard
prompt = hub.pull("rlm/rag-prompt")

def format_docs(docs):
    return "\n\n".join(doc.page_content for doc in docs)

rag_chain = (
    {"context": retriever | format_docs, "question": RunnablePassthr
    | prompt
    | llm
    | StrOutputParser()
)

# Test
```

```
query = "Qual è il tema principale dei documenti?"
risposta = rag_chain.invoke(query)
print(risposta)
```

Passo 6: Agente Agentico con LangGraph

Per un flusso più avanzato: L'agente decide se recuperare dal DB o rispondere direttamente.

Python

```
from langgraph.graph import StateGraph, END
from typing import TypedDict, Annotated
import operator
from langchain_core.messages import HumanMessage

# Stato
class AgentState(TypedDict):
    messages: Annotated[list, operator.add]

# Nodi
def retrieve(state):
    question = state["messages"][-1].content
    docs = retriever.invoke(question)
    return {"messages": [f"Contesto: {format_docs(docs)}"]}

def generate(state):
    response = llm.invoke(state["messages"])
    return {"messages": [response]}

# Grafo
graph = StateGraph(AgentState)
graph.add_node("retrieve", retrieve)
graph.add_node("generate", generate)

# Condizionale: Decide se recuperare
def decide(state):
    question = state["messages"][-1].content
    if "documento" in question.lower() or "pdf" in question.lower():
        return "retrieve"
    return "generate"

graph.set_entry_point("generate")
graph.add_conditional_edges("generate", decide, {"retrieve": "retrie
```

```
graph.add_edge("retrieve", "generate")

app = graph.compile()

# Test
inputs = {"messages": [HumanMessage(content="Riassumi i PDF")]}
for output in app.stream(inputs):
    print(output)
```

Passo 7: Visualizzare e Analizzare il DB in DBeaver

Poiché usiamo SQLiteVSS, il DB è un file SQLite standard (./rag_vector.db). DBeaver lo supporta nativamente.

1. Apri DBeaver.
2. Crea nuova connessione: **File > New > Database Connection > SQLite**.
3. Seleziona "File" come tipo, e specifica il percorso al file ./rag_vector.db (es. /percorso/assoluto/rag_vector.db).
4. Connetti: Vedrai la tabella rag_table (o il nome che hai scelto).
5. Querya:
    - SELECT * FROM rag_table LIMIT 10; per vedere righe (content = testo chunk, vector = BLOB binario embeddings).
    - Metadata in colonne separate o serializzate.
    - Nota: I vettori sono binari (non leggibili), ma puoi contare righe (SELECT COUNT(*) FROM rag_table;) o cercare testo (SELECT content FROM rag_table WHERE content LIKE '%parola%';).
6. Per ricerca vettoriale: Non direttamente in SQL (usa l'extension vss), ma per debug, è utile vedere i dati raw.

- **Se l'extension vss non carica**: Assicurati sqlite-vss sia installato; SQLiteVSS la carica automaticamente in codice.
- **Vantaggi con DBeaver**: Puoi backuppare, modificare metadata, o esportare dati. Per dataset grandi, filtra query.

Consigli Finali e Debug

- **Test con chat esportata**: Esporta questa chat come PDF, aggiungila ai file, e querya "tutorial langgraph pdf". Il retriever troverà chunk rilevanti basati su similarità semantica (embeddings).

- **Scalabilità**: Per dataset enormi, considera Qdrant o Pinecone. SQLiteVSS è ideale per locale/medio (milioni vettori).
- **Errori comuni**: Se embeddings falliscono, check API key. Per chunk vuoti, adatta splitter.
- **Documentazione**: https://python.langchain.com/docs/integrations/vectorstores/sqlitevss/ (per SQLiteVSS).

Esegui passo per passo! Se hai errori o vuoi modifiche (es. torna a Chroma), dimmi. 😊

∅

Quanto è Utile l'LLM nel RAG con DB, PDF o File di Testo?

Spiegazione Teorica e Pratica (con focus su Transformers)

Ciao! Ti spiego in modo chiaro e dettagliato **perché un Large Language Model (LLM) è fondamentale** in un sistema Retrieval-Augmented Generation (RAG) quando lavori con database, PDF o file di testo. Dividerò la spiegazione in teoria (come funzionano i transformers) e pratica (cosa fa esattamente l'LLM nel flusso RAG).

1. Cos'è il RAG e Perché Serve un LLM?

Il RAG combina due fasi:

- **Retrieval**: Cerchi informazioni rilevanti nei tuoi documenti (PDF, TXT, DB).
- **Augmented Generation**: Usi quelle informazioni per generare una risposta.

Senza LLM, potresti solo recuperare i chunk di testo più simili alla domanda (come fa un motore di ricerca classico).
Con l'LLM invece ottieni una risposta **coerente, naturale, contestualizzata e sintetica**, come se un esperto avesse letto i documenti e ti rispondesse in italiano perfetto.

L'LLM è quindi il "cervello" che trasforma informazioni grezze recuperate in una risposta utile.

2. Teoria: Come Funzionano i Transformers (la base degli LLM moderni)

Tutti gli LLM attuali (GPT, Llama, Mistral, Grok, ecc.) sono basati sull'architettura **Transformer** (introduzione nel paper "Attention is All You Need" del 2017).

Punti chiave dei Transformers rilevanti per il RAG:

- **Attention Mechanism**: Il modello "presta attenzione" a tutte le parole del testo contemporaneamente e capisce quali sono più importanti per il contesto. Esempio: Nella frase "Il cane che inseguiva il gatto è nero", l'attention collega "nero" a "cane", non a "gatto".
- **Comprensione Semantica Profonda**: Grazie a miliardi di parametri addestrati su enormi quantità di testo, il modello capisce significato, relazioni, logica, sinonimi, implicazioni.
- **Generazione Autoregressiva**: Predice la parola successiva dato il contesto precedente. Questo permette di generare testo fluido e coerente.
- **Context Window**: Può "vedere" migliaia di token contemporaneamente (es. GPT-4o ha ~128k token). Questo è cruciale nel RAG: gli passi contesto recuperato + domanda, e genera la risposta.

3. Ruolo Specifico dell'LLM nel Flusso RAG (con PDF/DB/TXT)

Immaginiamo il flusso tipico:

1. **Domanda utente**: "Qual è il riassunto del capitolo 3 del PDF sulle norme di sicurezza?"
2. **Retrieval** (senza LLM):
   - Embeddi la domanda → vettore numerico.
   - Cerchi nei vettori dei chunk del PDF/DB → recuperi i 4-5 chunk più simili (es. parti del capitolo 3).

   → A questo punto hai solo frammenti di testo grezzo, magari ripetitivi, non ordinati, con rumore.
3. **Augmentation & Generation** (qui entra l'LLM):
   L'LLM riceve un prompt strutturato tipo:

   text

```
 Usa solo le seguenti informazioni per rispondere alla domanda.
Se non sai, dì che non lo sai.

Contesto:
[Chunk 1: ...parla di estintori...]
[Chunk 2: ...procedure di evacuazione...]
[Chunk 3: ...responsabili sicurezza...]
[Chunk 4: ...norme aggiornate 2025...]
```

```
Domanda: Qual è il riassunto del capitolo 3 del PDF sulle norme

Risposta:
```

Cosa fa l'LLM grazie ai Transformers:

- **Comprende il contesto**: Usa l'attenzione per collegare informazioni tra i vari chunk (anche se sono frammentati).
- **Sintetizza**: Riassume in modo coerente, eliminando ripetizioni.
- **Ragiona**: Inferisce relazioni (es. "il responsabile deve coordinare l'evacuazione" anche se non è scritto esattamente così).
- **Mantiene fedeltà**: Se ben promptato, non inventa (riduce allucinazioni).
- **Risponde in linguaggio naturale**: In italiano fluido, con struttura logica.

Risultato: "Il capitolo 3 tratta le procedure di evacuazione, l'uso degli estintori e le responsabilità del personale addetto alla sicurezza. Le norme sono state aggiornate nel 2025..."

4. Quanto è Utile l'LLM? Confronto Con/Senza

Senza LLM (solo retrieval)	Con LLM (RAG completo)
Restituisci i chunk grezzi più simili	Risposta sintetica, chiara e contestualizzata
L'utente deve leggere e interpretare	L'utente riceve direttamente la risposta desiderata
Risposte frammentate, ripetitive	Risposta coerente, senza ripetizioni
Non capisce implicazioni o relazioni	Ragiona e collega concetti
Non gestisce domande complesse	Risponde a domande di sintesi, confronto, ragionamento
Alto effort per l'utente	Esperienza tipo "parlare con un esperto che ha letto tutto"

Conclusione: Il retrieval da solo è utile per ricerca keyword/esatta, ma **l'LLM rende il sistema veramente intelligente e user-friendly**.

5. Esempi Pratici di Utilità dell'LLM nel RAG

- **PDF tecnici lunghi**: Un manuale di 200 pagine. Retrieval trova i paragrafi rilevanti → LLM riassume in 3 frasi.
- **Database testuale**: Estraggi righe/record → LLM genera report naturale ("Nel 2025 i ricavi sono cresciuti del 15% grazie al settore X").
- **Chat aziendali su documenti interni**: L'LLM risponde a domande come "Qual è la policy sulle ferie?" citando il regolamento HR senza allucinare.

- **Domande complesse**: “Confronta la normativa 2023 con quella 2025” → LLM legge i chunk rilevanti e fa il confronto.

6. Limiti (e come mitigarli)

- **Allucinazioni**: Se il contesto non contiene l’info, l’LLM potrebbe inventare. → Soluzione: Prompt forte (“Rispondi solo basandoti sul contesto”) + temperature bassa (0 o 0.2).
- **Context window limit**: Non puoi passare troppi chunk. → Soluzione: Retrieval efficiente (k=4-8), chunk ben dimensionati, o tecniche avanzate (reranking, hypothetical questions).
- **Costo**: Ogni query chiama l’LLM. → Soluzione: Usa modelli piccoli/efficienti (gpt-4o-mini, Llama 3 8B locale).

Riassunto Finale

L’LLM (basato su Transformers) è **assolutamente essenziale** in un sistema RAG con PDF, file di testo o DB perché:

- Trasforma informazioni sparse e grezze in risposte utili e naturali.
- Aggiunge comprensione semantica, sintesi e ragionamento che un semplice motore di ricerca vettoriale non può avere.
- Rende l’intero sistema accessibile a utenti non tecnici.

Senza LLM avresti un “Google interno”, con l’LLM hai un “assistente esperto che ha letto tutti i tuoi documenti”.

Se vuoi, possiamo approfondire tecniche avanzate (es. HyDE, reranking, self-query retriever) o come ottimizzare il prompt per ridurre allucinazioni! 😊

Come Funzionano i Modelli Transformer: Spiegazione Dettagliata, Come Capirli e Come Assegnano Numeri alle Parole

Ciao! La tua domanda è super interessante e si collega direttamente a quanto stavamo discutendo sui sistemi RAG e sugli embeddings. Assumo che "dare i numeri alle persone" sia un refuso per "dare i numeri alle parole" (o simile), dato il contesto precedente su embeddings come "divisione di parole in numeri". Se intendi qualcos'altro, fammi sapere!

Procederò spiegando i **Transformer** (l'architettura alla base di modelli come GPT, BERT, ecc.), come funzionano, come capirli bene e come "assegnano numeri alle parole" tramite embeddings.

Dividerò la spiegazione in parti chiare: teoria di base, funzionamento passo-passo, consigli per capirli meglio, e focus sugli embeddings (i "numeri" alle parole).

1. Cos'è un Transformer e Perché è Rivoluzionario?

I Transformer sono un'architettura di rete neurale introdotta nel 2017 dal paper di Google "Attention is All You Need". Prima dei Transformer, i modelli per elaborare sequenze (come testo) usavano RNN (Reti Neurali Ricorrenti) o LSTM, che processavano il testo parola per parola in sequenza – lento e con problemi su testi lunghi (perdita di contesto).

I Transformer **risolvono questo** processando **tutto il testo in parallelo**, usando meccanismi di "attenzione" per catturare relazioni tra parole, anche distanti. Sono la base di quasi tutti gli LLM moderni (es. GPT-4, Grok, Llama).

- **Applicazioni**: Traduzione, generazione testo, riassunti, RAG (come nei nostri tutorial), visione (Vision Transformer), e persino audio/video.

2. Come Funzionano i Transformer: Spiegazione Passo-Passo

Un Transformer ha due parti principali: **Encoder** (per capire input) e **Decoder** (per generare output). Nei modelli come GPT, usano solo Decoder; in BERT, solo Encoder.

Architettura Generale:

- **Input**: Testo (es. "Il cane inseguiva il gatto").
- **Output**: Dipende dal task (es. traduzione, risposta).

Passi Interni (teoria semplificata, ma dettagliata):

1. **Tokenizzazione e Embeddings Iniziali** (Assegnare "numeri" alle parole – ne parliamo dopo):
    - Il testo viene diviso in token (parole o sub-parole, es. "inseguiva" → "insegu" + "iva").
    - Ogni token diventa un vettore numerico (embedding) di dimensioni fisse (es. 512 o 4096 valori float).
2. **Positional Encoding**:
    - I Transformer non hanno sequenza naturale (a differenza di RNN), quindi aggiungono "coordinate posizionali" ai vettori. Formula semplice: PE(pos, 2i) = sin(pos / 10000^(2i/d)), dove pos è posizione, d è dimensione embedding.

Questo dice al modello "questa parola è alla posizione 3", permettendo parallelo ma con ordine.

3. **Multi-Head Self-Attention** (Il Cuore: "Attention is All You Need"):
    - **Self-Attention**: Per ogni parola, calcola quanto "prestare attenzione" alle altre.
        - Crea 3 vettori da ogni embedding: Query (Q, "cosa cerco?"), Key (K, "cosa offro?"), Value (V, "info da passare").
        - Similarità: Score = (Q · K) / sqrt(d_k) (dot product normalizzato).
        - Softmax sugli score per pesi (es. 0.8 su "cane", 0.1 su "gatto").
        - Output: Somma ponderata dei V. Esempio: In "Il cane inseguiva il gatto", "inseguiva" presta attenzione a "cane" (soggetto) e "gatto" (oggetto).
    - **Multi-Head**: Ripete attention in parallelo (es. 8 heads), catturando relazioni diverse (sintattiche, semantiche). Poi concatena.
    - Vantaggio: Cattura contesto globale in un colpo solo, non sequenziale.
4. **Feed-Forward Neural Network**:
    - Dopo attention, ogni vettore passa through una rete fully-connected semplice (2 layer lineari con ReLU). Formula: FFN(x) = max(0, xW1 + b1) W2 + b2. Questo "raffina" le rappresentazioni.
5. **Layer Normalization e Residual Connections**:
    - Norm: Stabilizza training (media/varianza a 0/1).
    - Residual: Aggiunge input originale all'output (x + sublayer(x)), previene vanishing gradients.
6. **Stack di Layer**:
    - Encoder: 6-24 layer (attention + FFN).
    - Decoder: Simile, ma con masked attention (non vede futuro) + cross-attention (guarda output encoder).
7. **Output**:
    - Per generazione: Softmax su vettore finale per predire token successivo (autoregressivo: genera uno alla volta, usa output come input successivo).

Matematica Semplificata:

- Attention: Attention(Q, K, V) = softmax(QK^T / sqrt(d_k)) V
- Questo è efficiente (O(n^2) tempo, ma parallelo su GPU).

In sintesi: I Transformer "guardano" tutto insieme, pesano relazioni, e raffinano layer per layer.

3. Come Capire Bene Come Lavorano i Transformer: Consigli Pratici

Per capirli davvero, non basta teoria – serve visualizzare e sperimentare. Ecco come:

- **Visualizzazioni e Tool**:
  - **The Illustrated Transformer** (blog di Jay Alammar): Disegni interattivi passo-passo. Inizia qui – spiega attention con animazioni.
  - **BertViz**: Tool per visualizzare attention heads in modelli reali (es. su Hugging Face). Carica un modello e vedi come "presta attenzione".
  - **Tensor2Tensor o Hugging Face Transformers Library**: Prova codice Python. Esempio: Installa transformers e fai from transformers import BertModel; model = BertModel.from_pretrained('bert-base-uncased') – poi analizza output.
- **Risorse per Studiare**:
  - **Paper Originale**: Leggi "Attention is All You Need" (breve, 10 pagine).
  - **Corsi Gratuiti**:
    - "NLP Course" su Hugging Face (pratico, con codice).
    - "CS224n" di Stanford (lezioni YouTube su Transformer).
  - **Libri**: "Deep Learning" di Goodfellow (capitolo su seq models) o "Transformers for Natural Language Processing" (più hands-on).
- **Esercizi Pratici**:
  - Implementa un Transformer semplice da zero (usa PyTorch tutorial ufficiale).
  - Usa Colab: Cerca "Transformer from scratch Colab" – esegui e modifica parametri (es. num_heads=4 vs 8) per vedere effetti.
  - Debug: In un modello, stampa matrici di attention – vedrai pesi alti su parole correlate.
- **Trucchi per "Capire"**:
  - Pensa a attention come "ricerca interna": Query è una ricerca Google, Keys sono titoli risultati, Values sono contenuti.
  - Simula su carta: Prendi frase corta, calcola attention manuale (semplificato).
  - Confronta con umani: Noi "prestiamo attenzione" a parti rilevanti quando leggiamo – Transformer fa lo stesso, ma matematico.

Con pratica, passerai da "è magia" a "è un calcolo vettoriale efficiente".

4. Come Si Riesce a "Dare i Numeri alle Parole"? (Embeddings nei Transformer)

Questo è il primo passo: Assegnare "numeri" (vettori) alle parole per renderle processabili.

- **Teoria**: Le parole sono categoriche (non numeri), ma le reti neurali vogliono vettori. Embeddings mappano parole in spazio vettoriale continuo dove similarità = distanza bassa (cosine similarity).

    - Esempio: "Re" - "Uomo" + "Donna" ≈ "Regina" (famoso da Word2Vec, ma Transformer lo fa meglio).
- **Come Funziona nei Transformer**:
    1. **Vocabolario**: Modello ha un dizionario (es. 50k token).
    2. **Embedding Layer**: Una matrice learnable (es. 50k righe x 768 colonne). Ogni token ha una riga: vettore iniziale random.
    3. **Training**: Durante addestramento (su miliardi di testi), aggiorna vettori via backpropagation. Vettori simili finiscono vicini se appaiono in contesti simili (distribuzionale: "conosci una parola dalle sue compagnie").
        - Formula: Embedding(token_id) = embedding_matrix[token_id].
    4. **Contestualizzazione**: Nei Transformer, embeddings iniziali sono fissi, ma attention li raffina layer per layer → embeddings contestuali (diversi per contesto).
        - Es. "Banco" (pesce o mobili?) – Attention decide dal contesto.
- **Come "Si Riesce"**:
    - **Pre-Training**: Modelli come BERT imparano embeddings su task mascherati (predici parola nascosta).
    - **Dimensionalità**: 768-4096 dims catturano nuance (es. una dim per "genere", una per "sentimento").
    - **Visualizza**: Usa t-SNE per proiettare embeddings in 2D – vedrai cluster (animali insieme, colori insieme).

Nei nostri tutorial RAG: Usiamo embeddings pre-calcolati (es. OpenAI) per chunk, poi retrieval cerca simili.

Spero sia chiaro! Se vuoi esempi codice, diagrammi o chiarimenti (es. su attention heads), dimmi. 😊

Certo! Ti faccio tutto quello che hai chiesto, passo per passo, in modo chiaro e dettagliato. 😊

1. “Me lo fai questo?” – Il tutorial completo con SQLiteVSS

Sì, te lo faccio eccome! Ecco il **tutorial completo e pronto da copiare-incollare** per creare un sistema RAG con LangChain + LangGraph usando **PDF e file di testo**, con vector store

SQLiteVSS (perfetto perché lo puoi aprire direttamente con DBeaver e vedere tutto dentro un file .db SQLite).

Python

```python
# INSTALLAZIONE (una sola volta)
# pip install langchain langchain-community langchain-openai langgra

import os
from langchain_community.document_loaders import PyPDFLoader, TextLo
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_openai import OpenAIEmbeddings, ChatOpenAI
from langchain_community.vectorstores import SQLiteVSS
from langchain import hub
from langchain_core.runnables import RunnablePassthrough
from langchain_core.output_parsers import StrOutputParser
from langgraph.graph import StateGraph, END
from typing import TypedDict, Annotated
import operator
from langchain_core.messages import HumanMessage

# === 1. Impostazioni ===
os.environ["OPENAI_API_KEY"] = "sk-..."  # metti la tua chiave

# File da caricare (modifica con i tuoi percorsi)
pdf_files = ["pdf1.pdf", "pdf2.pdf", "pdf3.pdf"]      # i tuoi PDF
txt_files = ["txt1.txt", "txt2.txt"]                  # i tuoi TXT

# === 2. Caricamento documenti ===
all_docs = []

for pdf in pdf_files:
    loader = PyPDFLoader(pdf)
    all_docs.extend(loader.load())

for txt in txt_files:
    loader = TextLoader(txt, encoding="utf-8")
    all_docs.extend(loader.load())

print(f"Caricati {len(all_docs)} documenti.")

# === 3. Chunking ===
splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_ove
chunks = splitter.split_documents(all_docs)
print(f"Creati {len(chunks)} chunk.")

# === 4. Embeddings ===
```

```
embeddings = OpenAIEmbeddings(model="text-embedding-ada-002")

# === 5. Vector Store persistente con SQLiteVSS ===
DB_FILE = "rag.db"  # file SQLite che vedrai in DBeaver

# Prima volta: crea il DB
vectorstore = SQLiteVSS.from_documents(
    documents=chunks,
    embedding=embeddings,
    table="documenti_rag",
    db_file=DB_FILE
)

# Volte successive: carica il DB esistente (non ricalcola embeddings
# vectorstore = SQLiteVSS(embedding=embeddings, table="documenti_rag

retriever = vectorstore.as_retriever(search_kwargs={"k": 4})

# === 6. Catena RAG semplice ===
llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
prompt = hub.pull("rlm/rag-prompt")

def format_docs(docs):
    return "\n\n".join(doc.page_content for doc in docs)

rag_chain = (
    {"context": retriever | format_docs, "question": RunnablePassthr
    | prompt
    | llm
    | StrOutputParser()
)

# Test semplice
print(rag_chain.invoke("Di cosa parlano i documenti?"))

# === 7. Agente con LangGraph (opzionale ma potente) ===
class AgentState(TypedDict):
    messages: Annotated[list, operator.add]

def retrieve_node(state):
    question = state["messages"][-1].content
    docs = retriever.invoke(question)
    return {"messages": [f"Contesto recuperato:\n{format_docs(docs)}

def generate_node(state):
    response = llm.invoke(state["messages"])
    return {"messages": [response]}
```

```
graph = StateGraph(AgentState)
graph.add_node("retrieve", retrieve_node)
graph.add_node("generate", generate_node)

def router(state):
    last_msg = state["messages"][-1].content.lower()
    if any(word in last_msg for word in ["pdf", "documento", "conten
        return "retrieve"
    return "generate"

graph.set_entry_point("generate")
graph.add_conditional_edges("generate", router, {"retrieve": "retrie
graph.add_edge("retrieve", "generate")
app = graph.compile()

# Test agente
inputs = {"messages": [HumanMessage(content="Riassumi il contenuto d
for output in app.stream(inputs):
    print(output)
```

Apri rag.db con DBeaver → vedi la tabella documenti_rag con colonne content, vector (BLOB), metadata. Tutto persistente!

2. Come funzionano gli Embeddings? Spiegazione + Esempi

Gli **embeddings** sono il modo in cui il computer trasforma parole, frasi o paragrafi in **sequenze di numeri** (vettori) che catturano il **significato semantico**.

Come lavorano (teoria semplice)

1. Un modello neurale (tipo Transformer) viene addestrato su miliardi di frasi.
2. Impara a prevedere parole mancanti o relazioni tra parole.
3. Durante l'addestramento, crea una "mappa" interna dove parole con significato simile finiscono vicine nello spazio numerico.
4. Alla fine, per ogni parola/frase, estrae un vettore (es. 1536 numeri float tra -1 e 1).

Esempi concreti

Supponiamo vettori semplificati a 3 dimensioni (in realtà sono 768–1536):

Parola	Vettore semplificato	Distanza da "re"
re	[0.9, 0.8, -0.1]	0.0
regina	[0.85, 0.75, 0.1]	molto bassa
uomo	[0.2, 0.3, -0.5]	media
donna	[0.15, 0.25, 0.4]	media
gatto	[-0.7, -0.4, 0.8]	alta

Famosa proprietà:
vettore("re") − vettore("uomo") + vettore("donna") ≈ vettore("regina")

Questo significa che il modello ha capito il concetto di genere e ruolo!

Esempio pratico con frasi (non solo parole)

- "Il gatto beve latte" → vettore A
- "Il micio mangia il latte" → vettore B molto vicino ad A
- "La macchina corre veloce" → vettore C lontano da A e B

Quando fai retrieval in RAG:
Domanda → embedding → cerchi i chunk con embedding più vicini → recuperi quelli semanticamente simili.

3. Word2Vec vs BERT vs RoBERTa: Qual è meglio?

Modello	Anno	Tipo	Come impara	Qualità embeddings	Quando usarlo oggi
Word2Vec (Google)	2013	Statici (una parola = un vettore fisso)	Prevede parole vicine (skip-gram/CBOW)	Buoni per base	Quasi mai più (superato)
BERT (Google)	2018	Contestuali (stesso parola = vettori diversi a seconda del contesto)	Masked LM + Next Sentence Prediction	Ottimi	Ancora buono, ma migliorabile
RoBERTa (Facebook)	2019	Contestuali (miglioramento di BERT)	Solo Masked LM, più dati, training più lungo, no Next Sentence	**Migliori** tra i tre	Attualmente uno dei migliori open-source gratuiti per embeddings

Confronto pratico

Frase: "La banca è piena di soldi"

- **Word2Vec**: "banca" ha sempre lo stesso vettore → non distingue tra istituto finanziario e argine del fiume.
- **BERT/RoBERTa**: "banca" avrà vettore diverso:
  - In contesto finanziario → vicino a "soldi", "conto", "mutuo"
  - In contesto fiume → vicino a "argine", "acqua", "fiume"

Vincitore oggi per RAG:

- Per qualità/prestazioni: **text-embedding-ada-002** di OpenAI (il default nei nostri tutorial) o **RoBERTa** large se vuoi open-source.
- RoBERTa è meglio di BERT perché addestrato su più dati e con ottimizzazioni.
- Word2Vec è obsoleto per quasi tutti gli usi moderni.

Se vuoi embeddings gratuiti locali ottimi: usa sentence-transformers/all-MiniLM-L6-v2 o intfloat/multilingual-e5-large.

Dimmi se vuoi che ti prepari anche un esempio con embeddings locali gratuiti o se hai altri dubbi! Sei gentilissimo tu a seguire con tanta attenzione! 🚀

∅

Differenza tra Vectorization ed Embeddings: Spiegazione Chiara e Dettagliata

Ciao! La domanda è ottima perché questi due termini vengono spesso usati in modo confuso, specialmente nel contesto di NLP (Natural Language Processing), RAG e machine learning. In realtà **non sono la stessa cosa**, anche se sono strettamente collegati.

Vediamo la differenza in modo semplice, poi scendiamo nei dettagli con esempi.

Definizione Rapida

Termine	Significato Principale	Tipo di Rappresentazione	Contestuale?	Esempi Comuni
Vectorization	Processo generico di convertire dati non numerici (testo, immagini, categorie, ecc.) in vettori numerici	Qualsiasi tecnica di trasformazione in vettori	No (di solito)	Bag of Words, TF-IDF, One-Hot Encoding

Termine	Significato Principale	Tipo di Rappresentazione	Contestuale?	Esempi Comuni
Embeddings	Tipo specifico e avanzato di vectorization che produce vettori densi che catturano significato semantico	Vettori densi (pochi centinaia/migliaia di dimensioni) con relazioni semantiche	Sì (nei modelli moderni)	Word2Vec, GloVe, BERT, OpenAI embeddings

In breve:
Tutti gli **embeddings** sono una forma di **vectorization**,
ma **non tutta la vectorization** produce embeddings.

1. Vectorization (Vectorizzazione) – Concetto Generale

È semplicemente **qualsiasi tecnica che trasforma dati non strutturati o categorici in vettori numerici** affinché un algoritmo di machine learning possa elaborarli.

Esempi classici di vectorization (che NON sono embeddings):

- **One-Hot Encoding**: Parole: ["gatto", "cane", "gatto"] Vocabolario: ["cane", "gatto", "uccello"] → "gatto" diventa [0, 1, 0] Problema: vettori molto sparsi (tanti zeri), nessuna nozione di similarità (gatto e cane sono distanti quanto gatto e uccello).
- **Bag of Words (BoW)**: Frase: "Il gatto mangia il pesce" → conta quante volte appare ogni parola → vettore [1, 1, 1, 1, 0, ...] Ignora ordine e significato.
- **TF-IDF**: Come BoW ma pesa le parole rare di più. Ancora vettori molto sparsi (spesso 10.000+ dimensioni), nessuna semantica.

Queste tecniche sono **vectorization pura**: convertono testo in numeri, ma i vettori non catturano relazioni di significato.

2. Embeddings – Vectorization Avanzata con Significato Semantico

Gli embeddings sono vettori **densi** (es. 300–1536 dimensioni) in cui la **distanza tra vettori rappresenta similarità semantica**.

Caratteristiche principali:

- Vettori densi (pochi zeri).
- Parole simili hanno vettori vicini (misurati con cosine similarity).
- Catturano relazioni analogiche (es. re - uomo + donna ≈ regina).

Tipi di embeddings:

Tipo	Contestuali?	Esempio famoso	Come funziona
Statici	No	Word2Vec (2013), GloVe, FastText	Una parola ha sempre lo stesso vettore, indipendentemente dal contesto
Contestuali	Sì	BERT, RoBERTa, OpenAI ada-002	Lo stesso parola ha vettori diversi a seconda della frase (es. "banca" finanziaria vs fiume)

Esempi concreti di proprietà semantiche (solo negli embeddings, non nella vectorization classica):

- Word2Vec: vettore("Parigi") - vettore("Francia") + vettore("Italia") ≈ vettore("Roma")
- BERT / OpenAI embeddings: Frase 1: "La mela è sul tavolo" → embedding di "mela" vicino a "frutta" Frase 2: "La mela è un'azienda tecnologica" → embedding di "mela" vicino a "Apple", "iPhone", "Jobs"

Tabella di Confronto Diretto

Caratteristica	Vectorization Classica (BoW, TF-IDF, One-Hot)	Embeddings (Word2Vec, BERT, ecc.)
Dimensioni vettore	Molto alte (10k–100k)	Basse-medi (300–1536)
Densità	Sparsa (tanti zeri)	Densa (pochi zeri)
Cattura significato semantico?	No	Sì
Cattura contesto frase?	No	Sì (solo modelli contestuali)
Similarità tra parole simili	Nessuna (distanza casuale)	Alta (cosine similarity vicina a 1)
Uso tipico oggi in RAG	Quasi mai	Sempre (OpenAI embeddings, Sentence Transformers)

Nel Contesto RAG (come nei nostri tutorial)

- Usiamo **embeddings** (es. OpenAI text-embedding-ada-002) per convertire chunk di PDF e la domanda in vettori.
- Poi misuriamo la similarità tra il vettore della domanda e i vettori dei chunk.
- Questo funziona perché gli embeddings catturano il **significato**, non solo le parole esatte.

Se usassimo solo vectorization classica (es. TF-IDF):

- Troverebbe solo chunk con le stesse parole esatte.
- Perderebbe risposte semanticamente corrette ma con parole diverse (es. "felino" invece di "gatto").

Riassunto Finale

- **Vectorization** = processo generale: "trasformo qualcosa in vettore numerico".
- **Embeddings** = vectorization intelligente e moderna: "trasformo in vettore che rappresenta il significato e le relazioni semantiche".

Oggi, quando si parla di ricerca semantica, RAG, chatbot su documenti, si intendono quasi sempre **embeddings**, non la vectorization classica.

Se vuoi, ti posso fare un piccolo esempio codice che confronta TF-IDF vs embeddings su una frase! 😊

Spiegazione Chiarissima: Cos'è un LLM, Cos'è FFN, Encoder e Decoder nei Transformer

Ciao! Ti spiego tutto in modo semplice, passo per passo, partendo dalle basi fino ai dettagli tecnici, così capisci bene come funzionano i Large Language Models (LLM) come GPT, Grok, Llama, Claude, ecc.

1. Cos'è un LLM (Large Language Model)?

Un **LLM** è un modello di intelligenza artificiale addestrato su **enormi quantità di testo** (miliardi o trilioni di parole da internet, libri, articoli, codice, ecc.) per capire e generare linguaggio umano.

- **Large** = ha miliardi o centinaia di miliardi di parametri (i "pesi" della rete neurale).
- **Language Model** = prevede la parola successiva in una frase.

Esempio semplice:
Gli dai "Il cielo è azzurro e il sole..." → l'LLM prevede "brilla" o "splende".

Grazie a questo addestramento, un LLM può:

- Rispondere a domande
- Riassumere testi
- Tradurre
- Scrivere codice
- Ragionare

- Fare tutto quello che stiamo facendo noi ora in questa chat!

Tutti gli LLM moderni (dal 2018 in poi) sono basati sull'architettura **Transformer**.

2. Cos'è un Transformer? (La base di tutto)

È un tipo di rete neurale inventato nel 2017 che ha rivoluzionato l'NLP.
Ha due parti principali: **Encoder** e **Decoder**.

Immagina il Transformer come una "macchina per elaborare testo" con due moduli:

Modulo	Cosa fa	Usato in quali modelli?
Encoder	Legge e **capisce** il testo in input	BERT, RoBERTa (modelli per comprensione)
Decoder	**Genera** testo parola per parola	GPT, Llama, Grok, PaLM (modelli per generazione)
Entrambi	Traduzione, riassunto con input + output	T5, BART

3. Encoder vs Decoder: Differenze Principali

Caratteristica	Encoder	Decoder
Obiettivo	Comprendere il testo in ingresso	Generare testo nuovo
Direzione del flusso	Legge tutto il testo contemporaneamente	Genera una parola alla volta (autoregressivo)
Attention	Vede tutte le parole in entrambe le direzioni	Vede solo le parole già generate (masked attention)
Esempio di uso	Classificare sentiment, cercare risposte in un documento	Chat, scrivere storie, completare frasi
Modelli famosi	BERT, RoBERTa	GPT-3, GPT-4, Llama, Grok

- **Encoder** = "lettore attento": guarda l'intera frase avanti e indietro per capire il contesto.
- **Decoder** = "scrittore creativo": scrive una parola alla volta, usando solo ciò che ha già scritto + (eventualmente) ciò che ha capito dall'encoder.

4. Cos'è la FFN (Feed-Forward Network)?

La **FFN** (o Feed-Forward Neural Network) è una parte **molto importante** all'interno di ogni layer del Transformer (sia encoder che decoder).

È una piccola rete neurale classica (senza ricorsione o attention) che viene applicata **indipendentemente a ogni parola** dopo il meccanismo di attention.

Struttura tipica di una FFN:

text

```
FFN(x) = max(0, x @ W1 + b1) @ W2 + b2
```

- Due layer lineari (matrici di pesi W1, W2)
- Una attivazione ReLU (max(0, ...)) in mezzo
- Aumenta la capacità del modello di fare trasformazioni non lineari

Ruolo della FFN:

- Dopo che l'**attention** ha mescolato le informazioni tra le parole, la **FFN** "pensa" individualmente su ogni parola e raffina la sua rappresentazione.
- È come se dicesse: "Ora che ho visto il contesto, trasformo questa parola in qualcosa di più ricco e utile".

In ogni layer del Transformer trovi:

1. Multi-Head Self-Attention
2. **Feed-Forward Network (FFN)**
3. Layer Normalization + Residual Connections (per stabilizzare l'addestramento)

5. Come è Strutturato un Transformer (Riassunto Visivo)

Un layer tipico (sia encoder che decoder):

text

```
Input Embeddings + Positional Encoding
          ↓
   Multi-Head Self-Attention
          ↓
   Residual Connection + Layer Norm
          ↓
        Feed-Forward Network (FFN)
          ↓
   Residual Connection + Layer Norm
          ↓
Output del layer → va al layer successivo
```

- Ci sono tanti layer identici impilati (es. 12 in BERT base, 96 in GPT-4).
- Più layer = più capacità di capire contesti complessi.

6. Esempi Pratici per Capire

- **BERT** (solo Encoder): Gli dai una frase → capisce il significato profondo → utile per ricerca semantica, classificazione.
- **GPT** (solo Decoder): Gli dai "Scrivi una poesia sul mare" → genera parola per parola una poesia originale.
- **ChatGPT / Grok / tu ed io ora**: Usiamo un modello Decoder-only (tipo GPT o Llama). Quando tu scrivi, il modello:
    1. Legge tutta la conversazione (encoder-like internamente)
    2. Genera la mia risposta una parola alla volta (decoder)

Riassunto Finale

Termine	Significato semplice
LLM	Grande modello che capisce e genera testo umano
Transformer	Architettura alla base degli LLM moderni
Encoder	Parte che legge e comprende il testo
Decoder	Parte che genera testo nuovo, parola per parola
FFN	Piccola rete neurale dentro ogni layer che "raffina" le rappresentazioni delle parole

Se vuoi, posso farti un disegnino testuale più dettagliato o spiegarti come funziona l'attenzione passo per passo! 😊
Dimmi pure cosa vuoi approfondire di più!
