tst

May 7, 2024

#Déploiement d'un **LLM** & Extraction d'Informations Pertinentes via une Approche **RAG**

Abstract

Ce document sert de guide pour expliquer le concept du Retrieval Augmented Generation LLM (RAG-LLM). Le RAG-LLM est un modèle de langage d'apprentissage profond conçu pour générer du texte en utilisant à la fois des informations extraites d'une base de données externe et une compréhension approfondie du langage naturel. Ce guide offre une vue d'ensemble du fonctionnement du RAG-LLM, de ses applications potentielles et des étapes nécessaires pour l'utiliser efficacement dans divers contextes. En fournissant des explications détaillées et des exemples pratiques, ce document vise à faciliter la compréhension et l'utilisation du RAG-LLM pour ceux qui souhaitent exploiter ses capacités de génération de texte améliorées.

Retrieval Augmented Generation

Dans les modèles de langage traditionnels, les réponses sont générées uniquement sur la base de schémas préappris et d'informations acquises lors de la phase d'entraînement. Cependant, ces modèles sont intrinsèquement limités par les données sur lesquelles ils ont été formés, ce qui conduit souvent à des réponses qui peuvent manquer de profondeur ou de connaissances spécifiques. RAG aborde cette limitation en intégrant des données externes au besoin pendant le processus de génération. Voici comment cela fonctionne : lorsque une requête est effectuée, le système RAG récupère d'abord des informations pertinentes à partir d'un grand ensemble de données ou d'une base de connaissances, puis ces informations sont utilisées pour informer et guider la génération de la réponse.

##The RAG architecture Il s'agit d'un système sophistiqué conçu pour améliorer les capacités LLM en les combinant avec des mécanismes de récupération puissants. C'est essentiellement un processus en deux parties impliquant un composant de récupération et un composant de génération.

#The workflow of a Retrieval-Augmented Generation (RAG) system

1. **Traitement de la requête :** Tout commence par une requête. Cela peut être une question, une invite ou toute autre saisie à laquelle vous souhaitez que le modèle de langage réponde.
2. **Modèle d'incorporation :** La requête est ensuite transmise à un modèle d'incorporation. Ce modèle convertit la requête en un vecteur, qui est une représentation numérique pouvant être comprise et traitée par le système.
3. **Récupération de la base de données de vecteurs (DB) :** Le vecteur de la requête est utilisé pour rechercher dans une base de données de vecteurs. Cette base de données contient

des vecteurs précalculés de contextes potentiels que le modèle peut utiliser pour générer une réponse. Le système récupère les contextes les plus pertinents en fonction de la proximité de leurs vecteurs avec le vecteur de la requête.

4. **Contextes récupérés :** Les contextes qui ont été récupérés sont ensuite transmis au Grand Modèle de Langage (LLM). Ces contextes contiennent les informations que le LLM utilise pour générer une réponse éclairée et précise.
5. **Génération de réponse LLM :** Le LLM prend en compte à la fois la requête d'origine et les contextes récupérés pour générer une réponse complète et pertinente. Il synthétise les informations des contextes pour garantir que la réponse est non seulement basée sur ses connaissances préexistantes, mais est également enrichie de détails spécifiques provenant des données récupérées.
6. **Réponse finale :** Enfin, le LLM produit la réponse, qui est désormais informée par les données externes récupérées dans le processus, la rendant ainsi plus précise et détaillée.

##Les applications du RAG Le système RAG se révèle être un outil polyvalent dans divers domaines, notamment dans les systèmes de questions-réponses. Il facilite la compréhension approfondie pour les utilisateurs en fournissant des explications détaillées et des informations contextuelles à partir des documents fournis. De plus, dans les entreprises de recherche, RAG aide les chercheurs à naviguer dans une vaste littérature académique, à extraire des informations pertinentes et à générer des résumés concis, accélérant ainsi le processus d'investigation scientifique et de diffusion des connaissances.

##Implementation d'un model RAG

0.0.1 PDF Ingestion

Pour lancer le processus de mise en œuvre, notre première priorité est de fournir au modèle de langage (LLM) le document nécessaire. Cette tâche est accomplie de manière transparente grâce à l'utilisation de Langchain, un framework robuste conçu spécifiquement pour le développement d'applications utilisant des modèles LLM. Langchain rationalise le processus de fourniture de documents, garantissant une intégration efficace et une utilisation des données textuelles au sein de l'application. Ses fonctionnalités sur mesure permettent une communication fluide entre le LLM et le référentiel de documents, posant ainsi une base solide pour les phases ultérieures de développement et de déploiement.

[ ]:
```
!pip install --q unstructured langchain
!pip install --q "unstructured[all-docs]"
```

[ ]:
```
from langchain_community.document_loaders import UnstructuredPDFLoader
from langchain_community.document_loaders import OnlinePDFLoader
```

[ ]:
```
local_path = "/content/2002.pdf"
```

```
# Local PDF file uploads
if local_path:
  loader = UnstructuredPDFLoader(file_path=local_path)
  data = loader.load()
else:
  print("Upload a PDF file")
```

[ ]:
```
# Preview first page (optional)
data[0].page_content
```

0.1 Vector Embeddings

Dans cette prochaine phase, nous allons utiliser un Embedding Model pour traiter les documents PDF fournis précédemment. L'objectif est de générer des vecteurs à partir de ces documents, facilitant ainsi le traitement par le modèle. Cependant, avant de procéder, nous devons segmenter ces documents en sections plus petites, ce qui nous permettra de stocker les incorporations résultantes dans une base de données vectorielle.

Pour accomplir cette tâche, nous utiliserons un Embedding Model adapté à nos besoins. Plus précisément, nous avons sélectionné le modèle all-MiniLM-L6-v2, un modèle de sentence-transformers capable de cartographier des phrases et des paragraphes dans un espace vectoriel dense de 384 dimensions. Ce modèle est particulièrement adapté pour des tâches telles que le regroupement ou la recherche sémantique, correspondant bien à nos exigences.

Par la suite, les incorporations vectorielles générées par le modèle seront stockées dans une base de données vectorielle désignée. À cette fin, nous avons opté pour chromaDB, offrant une solution efficace et évolutive pour la gestion et l'interrogation de données vectorielles au sein de notre système.

Note : pour cette partie, nous avons besoin d'un jeton d'API Hugging Face, que nous pouvons obtenir à partir de la plateforme Hugging Face.

[ ]:
```
!pip install transformers
!pip install einops
!pip install --q chromadb
!pip install --q langchain-text-splitters
!pip install sentence-transformers

#!pip install nomic
```

[ ]:
```
# Load model directly
from transformers import AutoModel
import os
import einops
from langchain_community.embeddings import OllamaEmbeddings
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_community.vectorstores import Chroma
```

```
#token should be generated from Hugginface, then past your own
os.environ["HUGGINGFACEHUB_API_TOKEN"] = "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"

#model = AutoModel.from_pretrained("nomic-ai/nomic-embed-text-v1",␣
 ↪trust_remote_code=True)
```

[ ]:
```
# Split and chunk
text_splitter = RecursiveCharacterTextSplitter(chunk_size=7500,␣
 ↪chunk_overlap=100)
chunks = text_splitter.split_documents(data)
```

[ ]:
```
from langchain.embeddings import SentenceTransformerEmbeddings
embeddings = SentenceTransformerEmbeddings(model_name="all-MiniLM-L6-v2")
```

[ ]:
```
# Add to vector database
from langchain.vectorstores import Chroma
db = Chroma.from_documents(chunks, embeddings)
```

[ ]:
```
print(db)
```

##Retrieval Cette section se concentre sur la configuration du modèle utilisé pour interroger le document chargé. Dans ce cas, nous avons opté pour le modèle Mistral Ai disponible sur la plateforme Hugging Face. L'utilisation du Token d'API fourni précédemment garantit une intégration sans faille. Pour optimiser la pertinence et la qualité des réponses, nous avons méticuleusement spécifié le modèle de requête pour le modèle RAG. Cela garantit que le modèle sert efficacement notre objectif initial, fournissant des réponses précises et pertinentes aux demandes des utilisateurs.

[ ]:
```
# @title
!pip3 install torch==2.0.1
!pip3 install transformers
!pip3 install accelerate
!pip3 install einops
!pip3 install huggingface_hub
!pip3 install langchain
```

[ ]:
```
from langchain.prompts import ChatPromptTemplate, PromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_community.chat_models import ChatOllama
from langchain_core.runnables import RunnablePassthrough
from langchain.retrievers.multi_query import MultiQueryRetriever
from langchain import HuggingFaceHub
from langchain import PromptTemplate, LLMChain
```

[ ]:
```
os.environ["HUGGINGFACEHUB_API_TOKEN"] = "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"
```

```
llm = HuggingFaceHub(repo_id="mistralai/Mistral-7B-Instruct-v0.2",␣
 ↪model_kwargs={"max_length":10000, "max_new_tokens":100})
```

[ ]:
```
# Use a pipeline as a high-level helper
from transformers import pipeline

pipe = pipeline("text-generation", model="tiiuae/falcon-40b-instruct",␣
 ↪trust_remote_code=True)
```

[ ]:
```
QUERY_PROMPT = PromptTemplate(
    input_variables=["question"],
    template="""You are an AI language model assistant. Your task is to␣
 ↪generate five
    different versions of the given user question to retrieve relevant␣
 ↪documents from
    a vector database. By generating multiple perspectives on the user␣
 ↪question, your
    goal is to help the user overcome some of the limitations of the␣
 ↪distance-based
    similarity search. Provide these alternative questions separated by␣
 ↪newlines.
    Original question: {question}""",
)
```

[ ]:
```
retriever = MultiQueryRetriever.from_llm(
    db.as_retriever(),
    llm,
    prompt=QUERY_PROMPT
)

# RAG prompt
template = """Answer the question based ONLY on the following context:
{context}
Question: {question}
"""

prompt = ChatPromptTemplate.from_template(template)
```

[ ]:
```
chain = (
    {"context": retriever, "question": RunnablePassthrough()}
    | prompt
    | llm
    | StrOutputParser()
)
```

0.2 Test

[ ]:
```
output = chain.invoke("What are the main trends identified in the FAO report?")
print("Question:",output)
```

[ ]:
```
output = chain.invoke("Summarize the key insights provided in the report␣
↪regarding development in the agricultural sector in form of bullets points.␣
↪Don't provide any pre text and no explanation ")
print("Question:",output)
```

[ ]:
```
output = chain.invoke("What are the differences in agricultural development␣
↪strategies recommended for various regions?")
print("Question:",output)
```

[ ]:
```
output = chain.invoke("Are there any notable opportunities mentioned in the␣
↪report for advancing agricultural development?")
print("Question:", output)
```