Da Teoria à Prática: Modelos de IA Generativa com R e Python

Consultando e Extraindo Informações de PDFs com IA

Magno T. F. Severino

2024-08-08

Índice

code-block-bg: true

Organização e Apoio



Patrocínio



## 1 Introdução

### 1.1 Conceitos Básicos

### 1.2 Requisitos

R e RStudio Instalado (link para instalação)

Python Instalado (inserir tutorial de instalação através do r)

ChatGTP API: para usar a api do chatgpt, você deve [se cadastrar](https://platform.openai.com/). Além disso você deve adicionar créditos para conseguir utilizar a API da OpenAI. Entre [nesta página](https://platform.openai.com/settings/organization/billing/overview) e adicione. Sugiro adicionar US$ 1, que será mais do que suficiente para execução deste tutorial.

## 2 Conexão com a API da OpenAI

Na [área de API Keys](https://platform.openai.com/api-keys) você deve criar nova chave usando o botão *“Create new secret key”*. Salve essa chave, que será usada neste tutorial.

Use o código abaixo para salvar a chave como uma variável de ambiente do RStudio.

Sys.setenv(`OPENAI\_API\_KEY`= "COLE SUA CHAVE AQUI")

## 3 Configuração de ambiente para rodar Python no RStudio

library(reticulate)  
  
virtualenv\_create(envname = "langchain\_rag\_pdf",  
 packages = c( "langchain", "openai", "pypdf", "bs4",  
 "python-dotenv", "chromadb", "tiktoken",  
 "langchain-openai", "langchain-community"))  
  
reticulate::use\_virtualenv("langchain\_rag\_pdf")  
  
reticulate::py\_run\_string('  
print("Hello, world!")   
')  
  
api\_key\_for\_py <- r\_to\_py(Sys.getenv("OPENAI\_API\_KEY"))

## 4 Download e importação do arquivo em PDF

if(!(dir.exists("docs"))) {  
 dir.create("docs")  
}  
  
download.file("https://cran.r-project.org/web/packages/ggplot2/ggplot2.pdf",  
 destfile = "docs/ggplot2.pdf", mode = "wb")

from langchain.document\_loaders import PyPDFLoader  
  
my\_loader = PyPDFLoader('docs/ggplot2.pdf')  
print(type(my\_loader))  
  
all\_pages = my\_loader.load()  
  
print(type(all\_pages))   
  
print(len(all\_pages))

# carregar documento em R  
  
all\_pages\_in\_r <- py$all\_pages  
  
all\_pages\_in\_r[[1]]$metadata # metadatos do primeiro item  
nchar(all\_pages\_in\_r[[100]]$page\_content) # conta a quantidade de caracteres do centésimo item

## 5 Divisão do documento em pedaços

import openai  
openai.api\_key = r.api\_key\_for\_py   
from langchain.text\_splitter import RecursiveCharacterTextSplitter  
my\_doc\_splitter\_recursive = RecursiveCharacterTextSplitter()  
my\_split\_docs = my\_doc\_splitter\_recursive.split\_documents(all\_pages)

my\_split\_docs <- py$my\_split\_docs

## 6 Avaliação do custo da aplicação

my\_split\_docs <- py$my\_split\_docs  
length(my\_split\_docs)  
  
total\_tokens <- purrr::map\_int(my\_split\_docs,   
 ~ TheOpenAIR::count\_tokens(.x$page\_content)) |>   
 sum()

Atualmente (agosto de 2024), o custo do modelo **ada v2** usado para criação dos *embeddings* da OpenAI é de US$ 0,10 / 1M tokens. Como temos 153.172 tokens, o custo dessa etapa será de aproximadamente US$ 0,0153172.

## 7 Geração de *embeddings*

if(!dir.exists("docs/chroma\_db")) {  
 dir.create("docs/chroma\_db")  
}

import os  
os.environ["OPENAI\_API\_KEY"] = r.api\_key\_for\_py   
  
from langchain\_openai import OpenAIEmbeddings  
embed\_object = OpenAIEmbeddings()  
  
from langchain\_community.vectorstores import Chroma  
chroma\_store\_directory = "docs/chroma\_db"  
  
vectordb = Chroma.from\_documents(  
 documents=my\_split\_docs,  
 embedding=embed\_object,  
 persist\_directory=chroma\_store\_directory  
)  
  
# numero de embeddings criados  
print(vectordb.\_collection.count())

## 8 Geração de respostas únicas

### 8.1 Carregar embeddings

# carregar embeddings  
  
import openai  
openai.api\_key = r.api\_key\_for\_py   
from langchain\_community.embeddings import OpenAIEmbeddings  
embed\_object = OpenAIEmbeddings()  
  
from langchain\_community.vectorstores import Chroma  
chroma\_store\_directory = "docs/chroma\_db"  
vectordb = Chroma(persist\_directory=chroma\_store\_directory,   
 embedding\_function=embed\_object)

### 8.2 Definir qual modelo de linguagem será usado

# Set up the LLM you want to use, in this example OpenAI's gpt-3.5-turbo  
# from langchain.chat\_models import ChatOpenAI  
from langchain\_community.chat\_models import ChatOpenAI  
from langchain\_openai import ChatOpenAI  
the\_llm = ChatOpenAI(model\_name="gpt-3.5-turbo", temperature=0)  
  
# Create a chain using the RetrievalQA component  
from langchain.chains import RetrievalQA  
qa\_chain = RetrievalQA.from\_chain\_type(the\_llm, retriever=vectordb.as\_retriever())

### 8.3 Obtenção da resposta

my\_question = "How do you rotate text on the x-axis of a graph?"  
print(qa\_chain.invoke(my\_question))

## 9 Geração de respostas como em uma conversa

### 9.1 Junção da consulta com os pedaços do documento

from langchain\_core.prompts import ChatPromptTemplate, MessagesPlaceholder  
from langchain.chains import create\_retrieval\_chain  
from langchain.chains.combine\_documents import create\_stuff\_documents\_chain  
from langchain\_core.runnables.history import RunnableWithMessageHistory  
from langchain\_core.chat\_history import BaseChatMessageHistory  
from langchain\_community.chat\_message\_histories import ChatMessageHistory  
from langchain\_core.prompts import ChatPromptTemplate, MessagesPlaceholder  
  
retriever = vectordb.as\_retriever()  
  
llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0)  
  
### Contextualize question ###  
contextualize\_q\_system\_prompt = (  
 "Given a chat history and the latest user question "  
 "which might reference context in the chat history, "  
 "formulate a standalone question which can be understood "  
 "without the chat history. Do NOT answer the question, "  
 "just reformulate it if needed and otherwise return it as is."  
)  
  
contextualize\_q\_prompt = ChatPromptTemplate.from\_messages(  
 [  
 ("system", contextualize\_q\_system\_prompt),  
 MessagesPlaceholder("chat\_history"),  
 ("human", "{input}"),  
 ]  
)  
  
history\_aware\_retriever = create\_history\_aware\_retriever(  
 llm, retriever, contextualize\_q\_prompt  
)

### 9.2 Construção da resposta

system\_prompt = (  
 "You are an assistant for question-answering tasks. "  
 "Use the following pieces of retrieved context to answer "  
 "the question. If you don't know the answer, say that you "  
 "don't know. If the question is out of context of the "  
 "retrieved context, do not answerr and just say it is out of context.\n\n"  
 "{context}"  
)  
  
qa\_prompt = ChatPromptTemplate.from\_messages(  
 [  
 ("system", system\_prompt),  
 MessagesPlaceholder("chat\_history"),  
 ("human", "{input}"),  
 ]  
)  
  
question\_answer\_chain = create\_stuff\_documents\_chain(llm, qa\_prompt)  
  
rag\_chain = create\_retrieval\_chain(history\_aware\_retriever, question\_answer\_chain)

### 9.3 Geração da resposta através de uma conversa

store = {}  
  
def get\_session\_history(session\_id: str) -> BaseChatMessageHistory:  
 if session\_id not in store:  
 store[session\_id] = ChatMessageHistory()  
 return store[session\_id]  
  
conversational\_rag\_chain = RunnableWithMessageHistory(  
 rag\_chain,  
 get\_session\_history,  
 input\_messages\_key="input",  
 history\_messages\_key="chat\_history",  
 output\_messages\_key="answer",  
)  
  
def get\_answer(question, session\_id = "abc123"):  
 result = conversational\_rag\_chain.invoke(  
 {"input": question},  
 config={"configurable": {"session\_id": session\_id}},  
 )  
   
 return result

### 9.4 Usando o chat via Python

while True:  
 my\_question = input("Diga: ")  
   
 if my\_question.lower() == 'sair':  
 print("Encerrando o chatbot.")  
 break  
   
 result = get\_answer(my\_question, "abc123")  
   
 print(result["answer"])

### 9.5 Usando o chat via R

py\_run\_string('  
print(qa\_chain.run("How can I make a bar chart where the bars are steel blue? Answer in pt-br"))  
')  
  
py\_run\_string('  
print(qa\_chain.run("What is the capital of Australia?"))  
')