



SAPIENZA
UNIVERSITÀ DI ROMA

Tutorial: Programming in LLM

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica

Nicola De Siena

ID number 1926056

Academic Year 2023/2024

Thesis not yet defended

Tutorial: Programming in LLM

Sapienza University of Rome

© 2024 Nicola De Siena. All rights reserved

This thesis has been typeset by L^AT_EX and the Sapthesis class.

Author's email:

Abstract

Contents

1	Tutorial: Programming with LLMS	1
1.1	Prompt Engineering	2
1.1.1	GPT4All	3
1.1.2	Step 1: Choose a benchmark	7
1.1.3	Step 2: Create a connection to a database	7
1.1.4	Step 3: Create the prompt	8
1.2	RAG and Langchain	14
1.2.1	Simple RAG example	16
1.2.2	RAG test with PDF	19
1.3	Methods to evaluate the models	24
1.3.1	Word Embeddings script	24
1.3.2	F1-Score and Exact Match	31

List of Figures

1.1	GPT4All Logo	3
1.2	GPT4All interface to search for models and to download them	4
1.3	This the ouput that you should see	6
1.4	PostgreSql Interface from PgAdmin4	8
1.5	The results in our database	13
1.6	Output of this script	18
1.7	Example of output for this script	30
1.8	Example of F1-Score's visualization in PostgreSQL	33
1.9	Example of EM's visualization in PostgreSQL	33

Chapter 1

Tutorial: Programming with LLMS

Let's start introducing LLMs: **A large language model (LLM) is a type of artificial intelligence (AI) designed to understand and generate human language.** It's trained on massive amounts of text data, allowing it to perform tasks such as:

- **Translation:** Translating text from one language to another.
- **Writing different kinds of creative content:** Generating poems, code, scripts, musical pieces, email, letters, etc.
- **Answering your questions in an informative way:** Providing summaries of factual topics or creating stories.
- **Summarizing large passages of text:** Condensing long documents into shorter summaries.

In this tutorial, we will explore how to use GPT4All in combination with LangChain to perform prompt-based programming and Retrieval-Augmented Generation (RAG). We will start by explaining what a prompt is, how to use the GPT4All library, and then move on to basic and advanced examples of Question Answering using one and multiple models. Finally, we will cover how to implement the RAG architecture with LangChain library for more complex use cases.

Prerequisites to begin this tutorial:

- Know the Python programming language
- A minimum basis of knowledge about Large language models(LLMs) and Machine Learning

1.1 Prompt Engineering

Prompt engineering is the process of crafting effective prompts to guide large language models (LLMs) to generate the desired output. It involves carefully constructing instructions, questions, or requests that maximize the model's ability to produce relevant, informative, and creative responses.

Key elements of prompt engineering:

- **Clarity and specificity:** The prompt should be clear and unambiguous, avoiding vague or overly general language.
- **Contextual relevance:** Providing relevant context can help the LLM understand the desired output and avoid generating irrelevant or misleading information.
- **Instructional clarity:** Explicitly instructing the LLM on the desired format, tone, or style of the response can improve the quality of the output.
- **Creativity and experimentation:** Exploring different prompt formats and variations can help discover new and innovative ways to leverage the LLM's capabilities.

Effective prompt engineering is essential for getting the most out of LLMs and can be used for a wide range of applications, such as generating text, translating languages, writing different kinds of creative content, and answering your questions in an informative way.

1.1.1 GPT4All



Figure 1.1. GPT4All Logo

GPT4All is an open-source large language model (LLM) created by Nomic that can be run locally on your own device . This means you don't need an internet connection or a powerful GPU to use it. It's designed to be accessible and user-friendly, making it a great option for those who want to experiment with AI language models without the technical complexities.

Key features of GPT4All include:

- **Local execution:** Runs on your computer, providing privacy and control.
- **Open-source:** The model's code is publicly available, fostering community development and innovation.
- **Multiple models:** Offers different models with varying capabilities to suit different needs.
- **User-friendly interface:** Provides a simple and intuitive way to interact with the model.

Now let's see a simple example of prompt in Python with the help of GPT4All.

Step 1: Choose a model

After you downloaded GPT4All software for Desktop from the official website, you must choose the model for your prompt. There are a lot of choices on GPT4All and all the models are in .gguf format. The .gguf format is a specialized file format designed for storing and loading large language models (LLMs). It's optimized for efficiency and compatibility with various LLM implementations, making it a popular choice for models like those found in GPT4All.

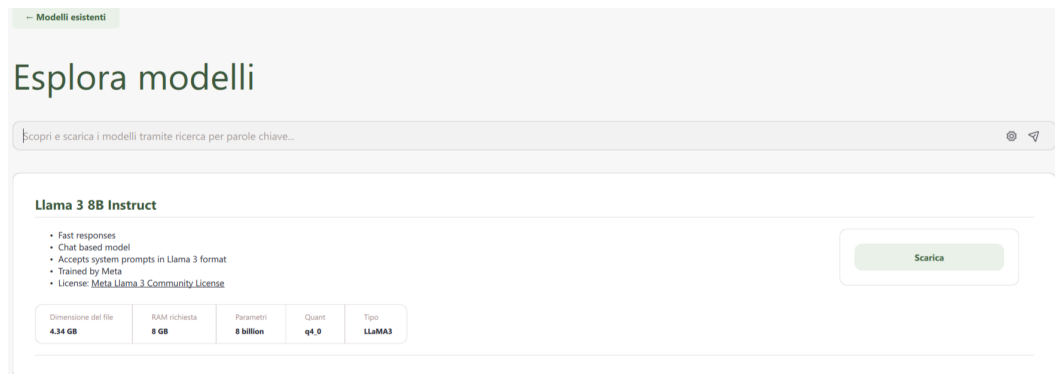


Figure 1.2. GPT4All interface to search for models and to download them

Each model has a description where the minimum requirements are indicated and whether it is free or not. The **B** in every model indicates the number of parameters that are essentially the "weights" and "biases" that the model learns during training. A larger number of parameters typically means a more complex model capable of handling more intricate tasks. So carefully choose the model best suited to your PC's requirements, given that a model with a rather high number of parameters will require more computation time if, for example, your RAM is low and your GPU is not powerful enough.

Step 2: Create virtual environment in Python and install packages

For this tutorial we are going to use Visual Code as text editor, but PyCharm, Google colab and Jupyter notebook are also a good choice. First of all let's create our python virtual environment:

- **Open the terminal and insert this command that leads to your path that will contain your project**

```
cd path_to_your_project
```

- **create the virtual environment**


```
python -m venv name_of_your_environment
```

- Then, to open every time your virtual environment, write this command in terminal:

```
name_of_your_environment\Scripts\activate
```

- Install the necessary packages with this command in the terminal:

```
pip install name_of_your_package
```

The packages are important to use the necessary libraries in the python script.

Please Note: Don't forget to use this command every time you need to add a new library in your project, because without the relative package you can't use it and you will get an error.

Step 3: Step to write the first script for a simple prompt

- Insert GPT4All library

```
from gpt4all import GPT4All
```

- Add the local path to your model

```
model_path = r'localpathtoyourmodeldownloadedfromgpt4all.gguf'
```

(For example, a local path could be C:\Users\nicol\gpt4all\resources\gpt4all-falcon-newbpe-q4_0.gguf)

- Initialize your model

```
gpt4all_model = GPT4All(model_path)
```

- Define the function to create the prompt and give it a context(in this case Formula 1)

```
def create_prompt(question):
    return f"Answer this question about Formula 1: {question}"
```

- Define the function to obtain the answers

```
def answer_question(question):prompt =
↪ create_prompt(question)response =
↪ gpt4all_model.generate(prompt)return response
```

- Let's give to the prompt some questions in input

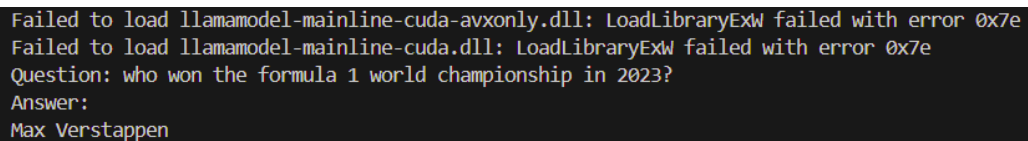
```
questions = ["who won the formula 1 world championship in  
→ 2023?", "what is the longest formula 1 track currently ?", "How  
→ many championships has Lewis Hamilton won?"]
```

- Let's execute this simple test

```
for question in questions: answer =  
→ answer_question(question) print(f"Question:  
→ {question}") print(f"Answer: {answer}\n")
```

- Start your python script in the terminal

```
python name_of_the_script.py
```



```
Failed to load llamamodel-mainline-cuda-avxonly.dll: LoadLibraryExW failed with error 0x7e  
Failed to load llamamodel-mainline-cuda.dll: LoadLibraryExW failed with error 0x7e  
Question: who won the formula 1 world championship in 2023?  
Answer:  
Max Verstappen
```

Figure 1.3. This the output that you should see

PLEASE NOTE : Even if you see these two errors "Failed to load Cuda,ecc." It's not a problem, because it's related to the fact that your GPU is not NVIDIA so it can't load some essential component to accelerate the computation of your model, so it can't use CUDA software. It happens if you're using LLAMA or similar models, but your script will still work without problems. (Obviously, if you have a NVIDIA GPU, you won't see this error message and all the process would be faster).

Prompt with Benchmark and more models

In the next section, we are going to show how to make a more complex prompt with more models with a huge number of questions as input and also we'll compare the answers of each model, saving the results in a database.

1.1.2 Step 1: Choose a benchmark

A **benchmark dataset** is a collection of data that is used to evaluate the performance of algorithms or models in a specific task. In our case, the benchmark contains questions about a topic/domain, the context for each question and the answers for each question. On the web you can find a lot of benchmarks suitable for question/answering tests, but in this case we chose the benchmark SQUAD, developed by Stanford university, which is a reading comprehension dataset, consisting of questions posed by crowdworkers on a set of Wikipedia articles, where the answer to every question is a segment of text, or *span*, from the corresponding reading passage, or the question might be unanswerable. Now go to this website <https://rajpurkar.github.io/SQuAD-explorer/> and download the **Dev Set 2.0 (4 Mb)**: it's a json file that contains all the question and answers that we're going to use as an input for our prompt.

1.1.3 Step 2: Create a connection to a database

For this test, we need to save the results in a database, instead of showing them on a terminal. For this example, we chose PostgreSQL, but MySQL, SQLite or other database are also a good choice. Before starting with python, you need to download PostgreSQL interface, known as pgAdmin from the official website <https://www.pgadmin.org/>. After the installation of the software, create your database with user and password.

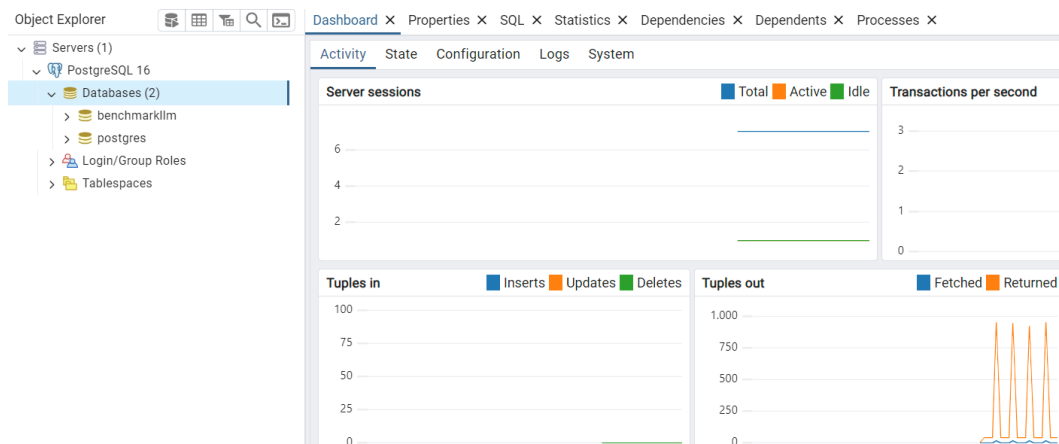


Figure 1.4. PostgreSQL Interface from PgAdmin4

- **First, install the package `psycopg2`** with command `pip install` and then add the `psycopg2` library that is fundamental to initialize a connection to a database in PostgreSQL

```
import psycopg2
```

- Define the function to connect to your database

```
def connecttodb():
    return psycopg2.connect(
        host="yourhost",
        database="yourdatabasename",
        user="yourusername",
        password="yourpassword" )
```

Please Note: usually "localhost" is a good choice for the hostname.

1.1.4 Step 3: Create the prompt

- Like we did before, add the `gpt4all` library and then load the local models

```
model_1 = GPT4All(r"localpath_to_your1st_model.gguf")
model_3 = GPT4All(r"localpath_to_your2nd_model.gguf")
model_2 = GPT4All(r"localpath_to_your3rd_model.gguf")
```

- Install the package `Json` and add the related library in the code that is essential to read `Json` file

```
import json
```

- Load the local benchmark SQUAD

```
with open(r"LocalPath_to_dev-v1.1.json") as f:
    squad_data = json.load(f)
```

- Extract the first n questions from the benchmark

```
squad_subset = []
for entry in squad_data['data']:
    for paragraph in entry['paragraphs']:
        for qa in paragraph['qas']:
            squad_subset.append({
                'context': paragraph['context'],
                'question': qa['question'],
                'answer': qa['answers'][0]['text']
            })
            if len(squad_subset) >= n:
                break
        if len(squad_subset) >= n:
            break
    if len(squad_subset) >= n:
        break
```

Please Note: For each entry, scroll through the paragraphs in entry['paragraphs']. Each paragraph contains text (paragraph['context']) and a list of questions and answers. Then scroll through each QA (qa) in paragraph['qas']. Each qa represents a question with one or more answers. To be more specific about the next lines, n could be 100,200 or even 1000, it's your choice. So, after adding an element to squad subset, check whether the list size has reached or exceeded n, the number of questions you want. If yes, it stops the current loop. The same control is applied in the outermost loops, so that when squad subset reaches size n, all loops stop, limiting the number of questions and answers selected.

- Define the function to ask questions to your models

```
def ask_question_gpt4all(model, question, context):  
    prompt = f"Context: {context}\nQuestion: {question}\nAnswer:"  
    response = model.generate(prompt)  
    return response
```

- **Create the table in PostgreSQL**(you can create it with a SQL script directly into pgAdmin with the query editor or you can write the query script in your code with a function in python in your code editor, both ways are correct). In this case, we are going to define the function to create the table:

```
def create_table(cur):  
    cur.execute("""  
        CREATE TABLE IF NOT EXISTS benchmark_results (  
            id SERIAL PRIMARY KEY,  
            question TEXT,  
            context TEXT,  
            ground_truth TEXT,  
            answer_orca TEXT,  
            answer_llama TEXT,  
            answer_falcon TEXT,  
            em_orca INTEGER,  
            f1_orca FLOAT,  
            em_llama INTEGER,  
            f1_llama FLOAT,  
            em_falcon INTEGER,  
            f1_falcon FLOAT  
        );  
    """)
```

- Define the function to save the results in the database

```
def save_to_db(cur, question, context, truth, answer_orca,
               answer_llama, answer_falcon):
    cur.execute("""
        INSERT INTO benchmark_results (question, context, ground_truth,
        answer_orca, answer_llama, answer_falcon)
        VALUES (%s, %s, %s, %s, %s, %s);
    """, (question, context, truth, answer_orca, answer_llama, answer_falcon))
```

- Start the connection to the database

```
conn = connect_to_db()
cur = conn.cursor()
```

- Now you need to iterate all the n questions from your local benchmark, save the questions in to the database and close the connection to it

```
try:
    # Iterate over the n local benchmark questions
    for idx, example in enumerate(squad_subset, 1):
        question = example['question']
        context = example['context']
        truth = example['answer']

        # Get answers from models
        answer_orca = ask_question_gpt4all(model_orca, question, context)
        answer_llama = ask_question_gpt4all(model_llama, question, context)
        answer_falcon = ask_question_gpt4all(model_falcon, question, context)

        # Save the results in to the db
        save_to_db(cur, question, context, truth, answer_orca, answer_llama, answer_falcon)

        # Confirm the insert after every iteration
        conn.commit()

        # Progressive printing on the terminal
        print(f"Saved {idx} results out of n.")

    print("Data successfully saved to database.")

except Exception as e:
    print(f"Error during the saving : {e}")
    conn.rollback()

finally:
    # Close the connection
    cur.close()
    conn.close()
```


ground_truth text	answer_orca text
Denver Broncos	The Denver Broncos
Carolina Panthers	The Carolina Panthers
Santa Clara, California	Super Bowl 50 took place at Levi's Stadium in Santa Clara, California.
Denver Broncos	The Denver Broncos won Super Bowl 50.
gold	Gold
"golden anniversary"	The theme of Super Bowl 50 was a "golden anniversary" celebration, with various gold-themed initiatives and temporarily suspending the tradition
February 7, 2016	The game was played on February 7, 2016.
American Football Conference	The AFC stands for American Football Conference, a division in the National Football League (NFL).
"golden anniversary"	The theme of Super Bowl 50 was a "golden anniversary" celebration, with various gold-themed initiatives and temporarily suspending the tradition
American Football Conference	American Football Conference
February 7, 2016	The Super Bowl was played on February 7, 2016.
Denver Broncos	The Denver Broncos won Super Bowl 50.
Levi's Stadium	Levi's Stadium
Santa Clara	San Francisco Bay Area
Super Bowl L	Super Bowl L
2015	The 2015 season
2015	The Denver Broncos secured their third Super Bowl title in 2016.
Santa Clara	San Francisco Bay Area
Levi's Stadium	Levi's Stadium
24-10	The final score of Super Bowl 50 was Denver Broncos 24, Carolina Panthers 10.
February 7, 2016	Super Bowl 50 took place on February 7th in the year 2016.

Figure 1.5. The results in our database

1.2 RAG and Langchain

RAG (Retrieval-Augmented Generation) and LangChain are two powerful tools in the field of AI, particularly in the realm of large language models (LLMs). While they serve different purposes, they often work synergistically to enhance the capabilities of AI systems.

RAG (Retrieval-Augmented Generation)

RAG is a technique that combines the strengths of information retrieval and generative AI. It involves two primary steps:

1. **Retrieval:** When presented with a query, a RAG system first searches through a vast knowledge base (like a document store or a database) to retrieve relevant information. This information can be in the form of text, code, or other structured data.
2. **Generation:** The retrieved information is then fed into a language model, which generates a response based on the query and the retrieved context. This response can be a summary, a translation, a creative text, or any other form of text generation.

LangChain

LangChain is a framework that facilitates the development of applications powered by large language models. It provides a modular and flexible approach to building LLM-based applications. Key features of LangChain include:

- **Modularity:** It breaks down LLM applications into reusable components like prompts, chains, and agents.
- **Flexibility:** It supports various LLM providers and allows for customization of the LLM interaction process.
- **Integration:** It integrates seamlessly with other tools and libraries, enabling the creation of complex AI systems.

RAG and LangChain are often used together to create more sophisticated and informative AI applications. Here's how they complement each other:

- **RAG as a Component:** RAG can be incorporated into LangChain as a retrieval component. This allows LangChain applications to access and leverage external knowledge sources to improve their responses.

- **Enhanced Response Generation:** By combining RAG's ability to retrieve relevant information with LangChain's flexible framework, developers can create AI systems that generate more accurate, informative, and contextually relevant responses.
- **Improved Factual Accuracy:** RAG helps mitigate the issue of hallucinations, which are instances where LLMs generate incorrect or misleading information. By grounding the LLM's responses in factual information, RAG can significantly improve the accuracy of the generated text.

In essence, RAG provides the knowledge base, while LangChain provides the framework to effectively utilize that knowledge. By working together, these two technologies enable the creation of more powerful and intelligent AI applications.

1.2.1 Simple RAG example

Choose your model, add the library GPT4All and load the model as we did before, but this time we add a new function.

- Load the model and then add a function to make the generation part easier

```
# Load your local model
model_path = r'C:\yourmodel.gguf'
gpt4all_model = GPT4All(model_path)

class LocalLLM:
    def __init__(self, model):
        self.model = model

    def __call__(self, prompt):
        response = self.model.generate(prompt)
        return response

llm = LocalLLM(gpt4all_model)
```

- Configure the Wikipedia retriever with an appropriate user agent

```
class WikipediaRetriever:
    def __init__(self):
self.wiki = Wikipedia(language='en',
user_agent= "RAG_project/1.0
(yourpersonalemail_here)")

    def retrieve(self, query):
        page = self.wiki.page(query)
        if page.exists():
            return page.summary
        else:
            return "No relevant information found."

retriever = WikipediaRetriever()
```

A Wikipedia Retriever class is defined to interact with Wikipedia via the wikipedia api API. This class sets a specific **user agent** for accessing Wikipedia and retrieves

the summary of a page related to the query, so don't forget to add it or the RAG won't work.

- **Configure the RAG chain**

```
class RetrievalQA:
    def __init__(self, llm, retriever):
        self.llm = llm
        self.retriever = retriever

    def __call__(self, query):
        context = self.retriever.retrieve(query)
        prompt = f"Context: {context}\n\nQuestion: {query}\nAnswer:"
        return self.llm(prompt)
```

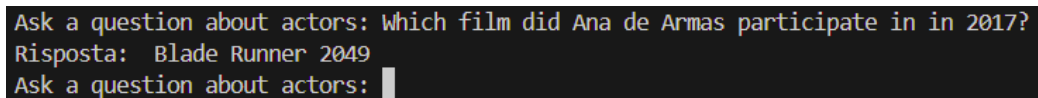
```
qa_chain = RetrievalQA(llm, retriever)
```

The **RetrievalQA** class combines the GPT4All model and the Wikipedia retriever. When called, it first searches Wikipedia for information using the retriever. The summary found is used as the context in the prompt to be provided to the GPT4All model, which answers the question based on that context. More specifically, in the constructor `init` we have two parameters: `llm` and `retriever`, the model and an object that retrieves context or relevant information based on the question provided. While the method call allows you to directly call the class instance as a function. It receives a `query` parameter, which represents the question you want to answer. Then we create an instance of the `RetrievalQA` class, called `qa chain`, passing `llm` and `retriever` as parameters. Now `qa chain` can be used as a function to handle questions and get answers based on context and language model.

- In the end execute the RAG chain you defined

```
def main():  
    while True:  
        query = input("Ask a question about (any context of your choice) : ")  
        if query.lower() in ['exit', 'quit']:  
            break  
        answer = qa_chain(query)  
        print(f"Answer: {answer}")  
  
if __name__ == "__main__":  
    main()
```

You can ask a question about any domain you like, there are no constraints about this choice. Also, in this case you'll ask the questions directly from the terminal, not from an input text written in the code like we did before in the prompt .

A terminal window with a dark background. The text displayed is: "Ask a question about actors: which film did Ana de Armas participate in in 2017?" followed by "Risposta: Blade Runner 2049" and then "Ask a question about actors:" followed by a cursor icon.

```
Ask a question about actors: which film did Ana de Armas participate in in 2017?  
Risposta: Blade Runner 2049  
Ask a question about actors: █
```

Figure 1.6. Output of this script

1.2.2 RAG test with PDF

The next test is more complex because we're going to give to our RAG model a local path that contains some PDFs the for the retrieval part and also a file Json containing a huge number of questions. Let's see how to proceed:

- First, you need to choose a common domain for all pdfs, in our case i chose some slides from the course of Software Engineering held by Prof. Massimo Mecella at Università la Sapienza di Roma. After that, add the usual libraries of Gpt4All, psycpg2 and define the functions to connect to the database, load the models and create a table in your database if you still don't have it.

Before you proceed with writing the rest of the script, create your json file with the questions and, note, it must be in a dictionary format like this:

```
[
{
  "id": 1,
  "text": "\nQuestion: What is a docker ?"
},
{
  "id": 2,
  "text": "\nQuestion: What are the key principles of DevOps ?"
}
]
```

- **Now add the langchain library and import PyPDFLoader:** this library allows you to upload PDF documents and split them into text. In this context, PDF files are uploaded to create an index based on the contents of those files, which will then be used to perform similarity searches (for example, to answer questions based on the contents of our PDFs). Also, import these other two libraries: HuggingFaceEmbeddings and FAISS. The first one provides the functionality to generate vector representations (embedding) of documents. In our case, the "sentence-transformers/all-MiniLM-L6-v2" template is used to transform documents into vectors representing their content. These vectors are then used to calculate the similarity to a asked question. The second one is a library for quickly searching for similarities between vectors. In our case, FAISS is used to index the uploaded PDF documents and allow you to search for those most similar to a specific question, in this case questions about Software Engineering. Now define the function to load PDFs from your local path and create an index:

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

def create_pdf_index(pdf_folder):
    loaders = []
    for file in os.listdir(pdf_folder):
        if file.endswith(".pdf"):
            loaders.append(PyPDFLoader(os.path.join(pdf_folder,
↪ file)))

    documents = []
    for loader in loaders:
        documents.extend(loader.load())

    embeddings =
↪ HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")
    index = FAISS.from_documents(documents, embeddings)
    return index
```



```
\paragraph{\subsection{\section{pdf_index =
↪ create_pdf_index(r"C:localpath_toyour_PDFs")}}}
```

in this code we create a list called loaders to store PDF loads. The function iterates over the files in the pdf folder, adding an instance of PyPDFLoader to loaders for each PDF file found (checked via the .pdf extension). Then we create a documents list to contain the texts extracted from each PDF. For each loader in the loaders list, call the load() method which loads and converts the contents of the PDF to text, adding it to documents. Then we create an instance **embeddings** with a pre-trained sentence-transformers model that will convert documents into numeric representations. We have **FAISS.fromdocuments** that converts documents into numeric vectors with embeddings and stores them in a FAISS index, which allows quick searches by similarity.

- **Load your file Json containing questions**

```
with open(r"C:Your_File_with_questions.json") as f:
questions = json.load(f)
```

- Now we need to load the **tokenizer**, but let's explain first why we need it: tokenizer has the task of managing the number of tokens (i.e. fragments of text, such as words or subwords) that can be processed by the language models. Our code uses the tokenizer to process text extracted from PDF documents (which is the relevant context for the question). After a similarity search is performed between the question and the content of the PDFs, the tokenizer converts the resulting text into a sequence of tokens. This is important because language models are limited in the number of tokens they can process in a single request. This operation is necessary because, if the context is too long, the model would not be able to process it entirely together with the question.

The bigger model you choose as a tokenizer, the better.

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("model_you_choose_as_tokenizer")
```

- **Define the function to ask questions**

```
def ask_question_to_models(models, question, pdf_index):

context = pdf_index.similarity_search(question, k=3)
```

```

context_text = " ".join([doc.page_content for doc in context])

# Tokenize context to check length
context_tokens = tokenizer(context_text)["input_ids"]

# Limit the number of total tokens (for example, 2048 - question
↪ length)
max_context_tokens = 1800 # Leave a margin for the question
if len(context_tokens) > max_context_tokens:
    # Crop the context to not exceed 1800 tokens
    context_text =
    ↪ tokenizer.decode(context_tokens[:max_context_tokens])

# Ask questions to every model
responses = {}
for model_name, model in models.items():
    prompt = f"Context: {context_text}\nQuestion:
    ↪ {question}\nAnswer:"

    # Use the generate method to get the response
    response = model.generate(prompt)

    # Tests whether the response is a string or a dictionary, if
    ↪ it is a string you'll get an error
    if isinstance(response, str):
        responses[model_name] = response
    elif isinstance(response, dict) and "choices" in response:
        responses[model_name] = response["choices"][0]["text"]
    else:
        responses[model_name] = "Error: Unexpected response
        ↪ format"

    print(f"Model: {model_name} - Question: {question[:30]}... -
    ↪ Response: {responses[model_name][:50]} ")

return responses

```

- Save questions and answers in to the database

```
for question in questions:
    q_id = question["id"]
    q_text = question["text"]
    print(f"Processing Question ID: {q_id} - {q_text[:50]}...")

    responses = ask_question_to_models(models, q_text, pdf_index)

    result = {
        "question_id": q_id,
        "question_text": q_text,
        "answer_llama": responses.get("Llama", ""),
        "answer_orca": responses.get("Orca", ""),
        "answer_falcon": responses.get("Falcon", "")
    }

    save_to_db(result)
    print(f"Saved response for question ID {q_id} to the database.")
```

1.3 Methods to evaluate the models

Eventually, you can add some other variables and methods in your tests, both Prompt and RAG, to evaluate your models. For example, you can add time counters to see which model was faster or you can add machine learning metrics like F1-Score or Exact Match. Another method to evaluate is the word embeddings: word embeddings are vector representations of words that capture their semantic meaning and they are obtained through deep learning models or techniques such as Transformer-based models and we'll see in example in the next page. The idea is that words that are similar in context and meaning have vector representations that are close in space. For example, the words "re" and "regina" will be closer to each other in vector space than "re" and "gatto". The closer the vectors are, the greater the semantic similarity between the words. Now let's see how to use it.

1.3.1 Word Embeddings script

- First, import the library sentence transformers to load a pre-trained model to convert texts to vector embeddings

```
embedding_model = SentenceTransformer('sentence-transformers_model_ofyourchoice')
```

- Import psycopg2, define a function to connect to your db(the one that contains the answers of your models to the benchmark and also the right answers for every questions) and retrieve the informations you need

```
def retrieve_responses():
    conn = psycopg2.connect(
        host="yourhost",
        database="yourdb",
        user="youruser",
        password="whatever"
    )
    cursor = conn.cursor()

    # Retrieve all answers from the database, including the truth
    cursor.execute("SELECT id, question, ground_truth,
↪ answer_1stmodel, answer_2ndmodel, answer_3rdmodel FROM
↪ benchmark_results_yourdb")
    results = cursor.fetchall()

    cursor.close()
```

```
conn.close()
```

```
return results
```

- Import numpy to measure the length of vectors and sklearn to calculate cosine similarity between vectors, then define a function to compute the embeddings and to compare the answers

```
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np
```

```
def compare_responses_with_embeddings(results):
    # Counters to track how many times each pattern is the best
    best_model_correctness_count = {
        "1stmodel": 0,
        "2ndmodel": 0,
        "3rdmodel": 0
    }
```

```
    best_model_semantic_count = {
        "1stmodel": 0,
        "2ndmodel": 0,
        "3rdmodel": 0
    }
```

```
comparison = []
```

```
for row in results:
    q_id, q_text, ground_truth, answer_1stmodel, answer_2ndmodel,
    ↪ answer_3rdmodel = row

    # Compute embeddings for each answer and for the ground
    ↪ truth
    embedding_1st = embedding_model.encode(answer_1stmodel)
    embedding_2nd = embedding_model.encode(answer_2ndmodel)
    embedding_3rd = embedding_model.encode(answer_3rdmodel)
    embedding_ground_truth =
    ↪ embedding_model.encode(ground_truth)
```

```

# Measure the length of vectors (semantic richness)
length_1stmodel = np.linalg.norm(embedding_1stmodel)
length_2ndmodel = np.linalg.norm(embedding_2ndmodel)
length_3rdmodel = np.linalg.norm(embedding_3rdmodel)

# Check how similar the answers are to the ground truth
↳ using cosine similarity
similarity_1stmodel = cosine_similarity([embedding_1stmodel],
↳ [embedding_ground_truth])[0][0]
similarity_2ndmodel = cosine_similarity([embedding_2ndmodel],
↳ [embedding_ground_truth])[0][0]
similarity_3rdmodel = cosine_similarity([embedding_3rdmodel],
↳ [embedding_ground_truth])[0][0]

# Set a threshold to consider an answer "correct"
similarity_threshold = 0.8 #you can use any value you want
↳ for your test

# Check if the answers are correct with respect to the
↳ similarity threshold
correct_models = []
if similarity_1stmodel >= similarity_threshold:
    correct_models.append("1stmodel")
    best_model_correctness_count["1stmodel"] += 1

if similarity_2ndmodel >= similarity_threshold:
    correct_models.append("2ndmodel")
    best_model_correctness_count["2ndmodel"] += 1

if similarity_3rdmodel >= similarity_threshold:
    correct_models.append("3rdmodel")
    best_model_correctness_count["3rdmodel"] += 1

# Among the correct models, determine which has the best
↳ semantic richness
if correct_models:
    best_model = None
    max_length = 0

```

```

if "1stmodel" in correct_models and length_llama >
↳ max_length:
    best_model = "1stmodel"
    max_length = length_1stmodel

if "2ndmodel" in correct_models and length_2ndmodel >
↳ max_length:
    best_model = "2ndmodel"
    max_length = length_2ndmodel

if "3rdmodel" in correct_models and length_3rdmodel >
↳ max_length:
    best_model = "3rdmodel"
    max_length = length_3rdmodel

if best_model:
    best_model_semantic_count[best_model] += 1
    print(f"Question ID: {q_id} - Best Model by Semantic
↳ Richness (among correct ones): {best_model}")
    print(f"1stmodel Length: {length_1stmodel}, 2ndmodel
↳ Length: {length_2ndmodel}, 3rdmodel Length:
↳ {length_3rdmodel}")
    print(f"1stmodel Similarity:
↳ {similarity_1stmodel:.2f}, 2ndmodel Similarity:
↳ {similarity_2ndmodel:.2f}, 3rdmodel Similarity:
↳ {similarity_3rdmodel:.2f}")
else:
    print(f"No correct model for Question ID {q_id}")

print()

```

Print the final results: which model answered correctly the most

↳ times

```

print("\nCorrectness Results:")
for model, count in best_model_correctness_count.items():
    print(f"{model} was correct {count} times.")

```

```
# Print the final results: which model has the best semantics
↪ among the correct ones
print("\nSemantic Richness Results (among correct models):")
for model, count in best_model_semantic_count.items():
    print(f"{model} was the best model by semantic richness
    ↪ {count} times.")

return comparison
```


- Retrieve the answers and compare them with embeddings and cosine similarity

```
results = retrieve_responses()

comparison = compare_responses_with_embeddings(results)
```

The function above compare response with embeddings starts by creating two dictionaries, best model correctness count and best model semantic count, to track the number of correct answers and the semantic quality of the answers for each model. Initializes a comparison list that will be used to store the results of each response and for each row in results, which represents a question and the models' answers the function computes embeddings for each model answer and for the correct answer (ground truth) via embedding `model.encode()`. For each response embedding, the function measures the length of the vector (representing the semantic richness of the response) using the norm of each embedding. Then the function calculates the cosine similarity between each answer and the correct answer (ground truth) using cosine similarity, which returns a value between 0 and 1 to indicate how semantically similar each answer is to the correct answer and a similarity threshold is set (similarity threshold = 0.8) to consider an answer "correct". The function checks for each model whether its similarity is greater than or equal to this threshold. If so, it adds the model to a correct models list and increments the best model correctness count counter for that model. If there are models with answers considered correct, the function selects the model with the greatest semantic richness (i.e., with the highest embedding length). Increment the best model semantic count counter for the model with the greatest semantic richness. At the end of the loop, the function prints how many correct results each model obtained (best model correctness count) and which was the best in terms of semantic richness among the correct models (best model semantic count).

```
Question ID: 98 - Best Model by Semantic Richness (among correct ones): Orca
Llama Length: 1.0, Orca Length: 1.0, Falcon Length: 1.0
Llama Similarity: 0.58, Orca Similarity: 1.00, Falcon Similarity: 0.98

Question ID: 100 - Best Model by Semantic Richness (among correct ones): Orca
Llama Length: 1.0, Orca Length: 1.0, Falcon Length: 0.9999999403953552
Llama Similarity: 0.27, Orca Similarity: 1.00, Falcon Similarity: 0.42

Correctness Results:
Llama was correct 0 times.
Orca was correct 48 times.
Falcon was correct 3 times.

Semantic Richness Results (among correct models):
Llama was the best model by semantic richness 0 times.
Orca was the best model by semantic richness 48 times.
Falcon was the best model by semantic richness 1 times.
```

Figure 1.7. Example of output for this script

1.3.2 F1-Score and Exact Match

Let's see how to use these two metrics in our test:

Exact Match: It is a search term matching method that requires a user's search query to be an exact match of a keyword or phrase in the content being searched. This means that the search query must appear in the exact same order and with the exact same spelling and capitalization as the keyword or phrase.

```
def exact_match(pred, truth):  
    return int(pred.strip().lower() == truth.strip().lower())
```

This exact match function checks if two text strings, **pred** (predicted text) and **truth** (actual text), match exactly, ignoring case and surrounding whitespace. With **strip()**, it removes any leading or trailing whitespace from both **pred** and **truth** and with **lower()** it converts both texts to lowercase, so the comparison ignores any differences in letter casing. Then it checks if the normalized **pred** and **truth** texts are equal and if they are exactly the same (after stripping whitespace and converting to lowercase), the expression evaluates to **True** otherwise it evaluates to **False**.

F1 Score: it's a metric that combines precision and recall into a single value. It is often used in machine learning, particularly in classification tasks, to evaluate the performance of a model. The components of F1-score are:
Precision: Measures the proportion of positive predictions that are actually correct. A high precision means that the model is good at avoiding false positives.
Recall: Measures the proportion of actual positive cases that the model correctly predicted. A high recall means that the model is good at avoiding false negatives.

A perfect F1-score of 1 indicates that the model has achieved both perfect precision and recall. A lower F1-score indicates that the model is either missing some positive cases or predicting false positives.

```
def f1(pred, truth):
    pred_tokens = pred.split()
    truth_tokens = truth.split()
    common = set(pred_tokens) & set(truth_tokens)
    if len(common) == 0:
        return 0.0
    precision = len(common) / len(pred_tokens)
    recall = len(common) / len(truth_tokens)
    return 2 * (precision * recall) / (precision + recall)
```

In details, **pred.split** splits the predicted text **pred** into a list of words (tokens) based on spaces, while **truth.split** does the same for the truth text. Then in **predtokens** and **truthtokens** finds the intersection between the sets, meaning only the tokens present in both sets remain in common. If there's no overlap, **common** will be an empty set. Checks if common is empty by testing if `len(common) == 0`. If there are no common tokens, it immediately returns 0.0 since both precision and recall would be zero, leading to an F1-score of zero. Then calculate **precision** and **recall** and after compute F1-score.

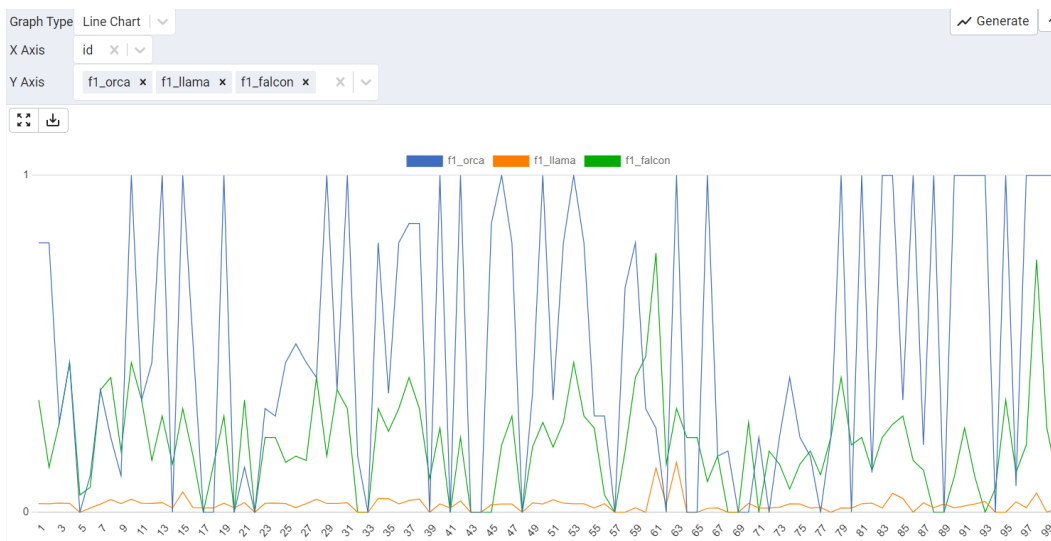


Figure 1.8. Example of F1-Score's visualization in PostgreSQL

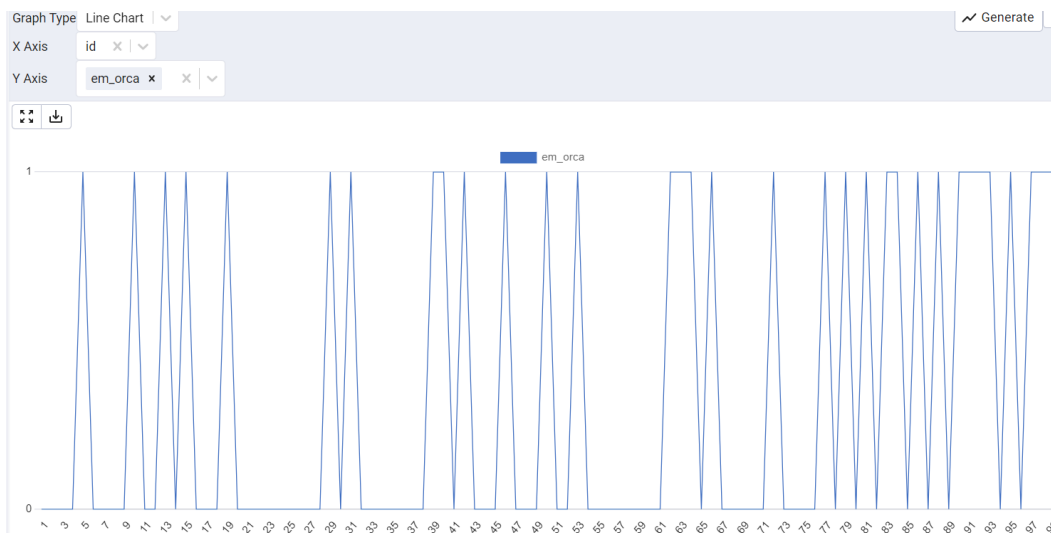


Figure 1.9. Example of EM's visualization in PostgreSQL