Member-only story

Running Llama 2 on CPU Inference Locally for Document Q&A

Clearly explained guide for running quantized open-source LLM applications on CPUs using Llama 2, C Transformers, GGML, and LangChain

Kenneth Leung · Follow
Published in Towards Data Science
11 min read · Jul 18

Listen Share More

Photo by NOAA on Unsplash

Third-party commercial large language model (LLM) providers like OpenAI's GPT4 have democratized LLM use via simple API calls. However, teams may still require self-managed or private deployment for model inference within enterprise perimeters due to various reasons around data privacy and compliance.

The proliferation of open-source LLMs has fortunately opened up a vast range of options for us, thus reducing our reliance on these third-party providers.

When we host open-source models locally on-premise or in the cloud, the dedicated compute capacity becomes a key consideration. While GPU instances may seem the most convenient choice, the costs can easily spiral out of control.

In this easy-to-follow guide, we will discover how to run quantized versions of open-source LLMs on local CPU inference for retrieval-augmented generation (aka document Q&A) in Python. In particular, we will leverage the latest, highly-performant Llama 2 chat model in this project.

Contents

The accompanying GitHub repo for this article can be found **here**.

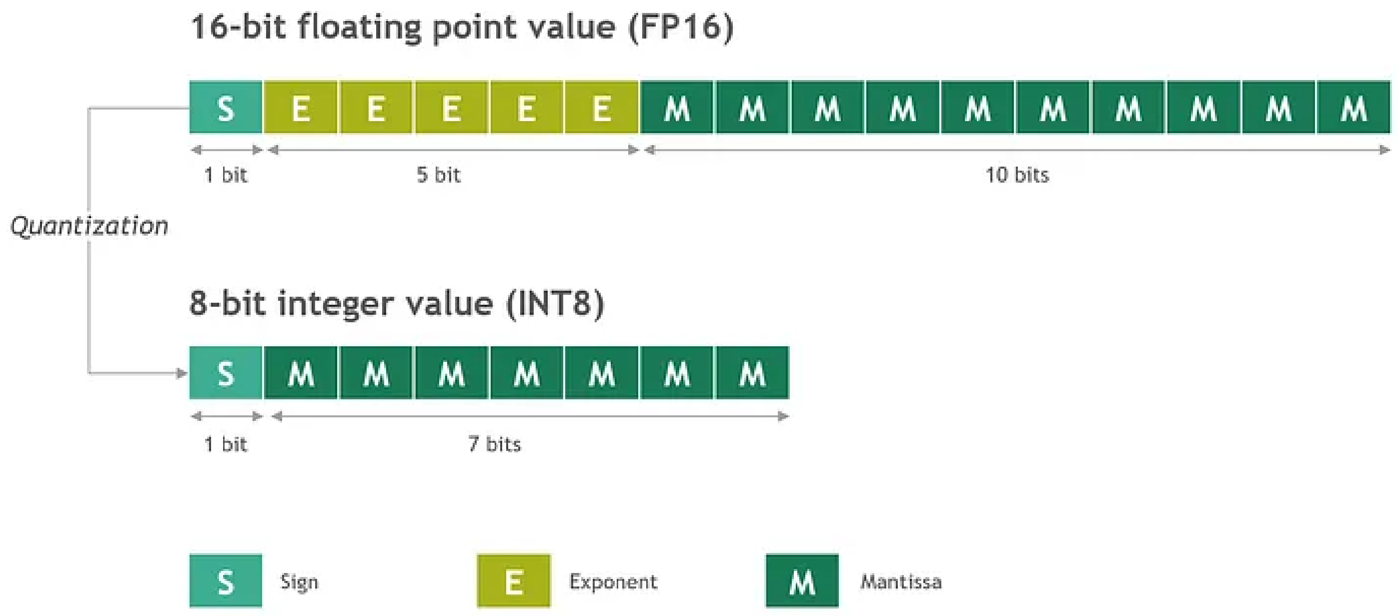
(1) Quick Primer on Quantization

LLMs have demonstrated excellent capabilities but are known to be compute- and memory-intensive. To manage their downside, we can use quantization to compress these models to reduce the memory footprint and accelerate computational inference while maintaining model performance.

Quantization is the technique of reducing the number of bits used to represent a number or value. In the context of LLMs, it involves reducing the precision of the model's parameters by storing the weights in lower-precision data types.

Since it reduces model size, quantization is beneficial for deploying models on resource-constrained devices like CPUs or embedded systems.

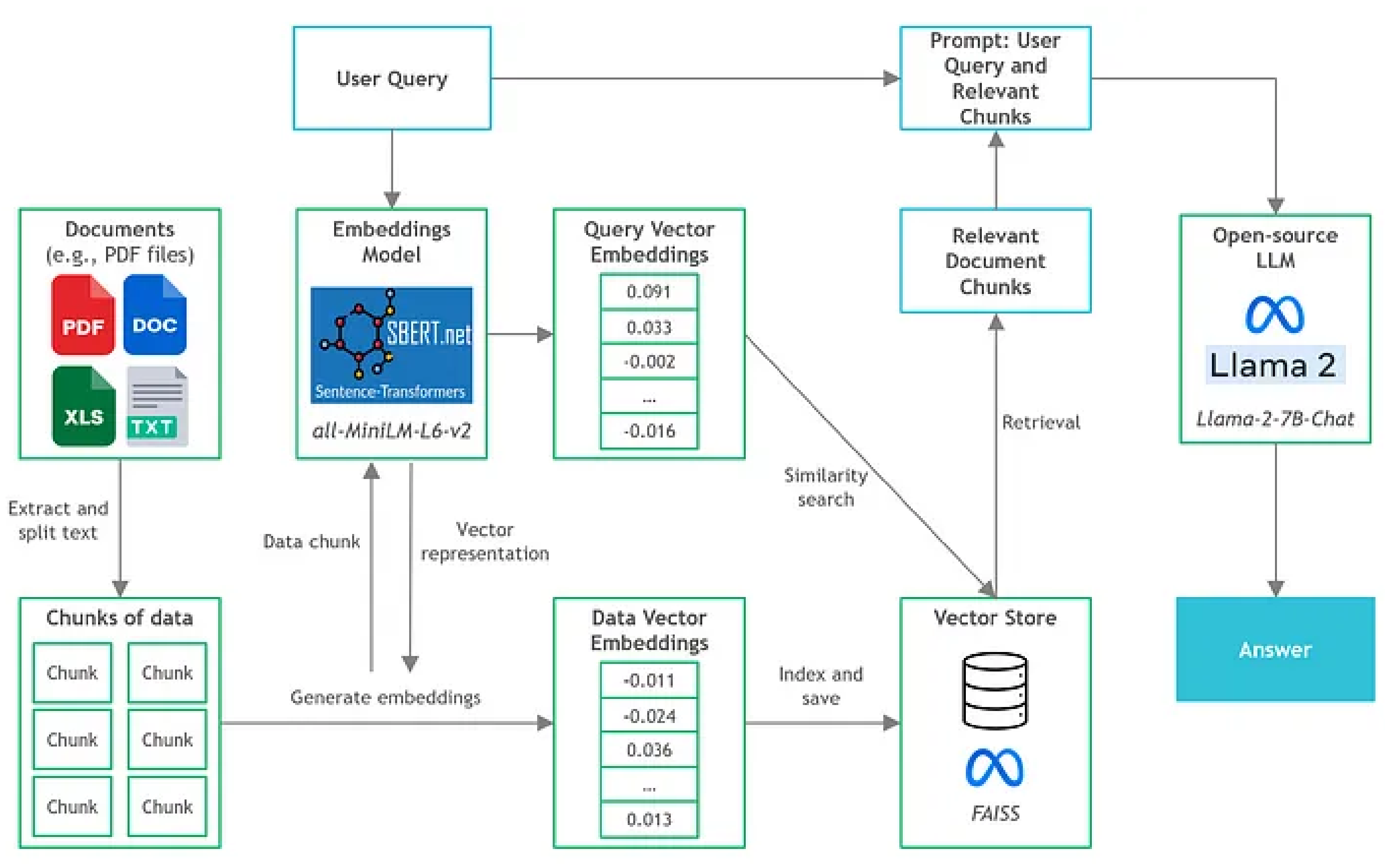
A common method is to quantize model weights from their original 16-bit floating-point values to lower precision ones like 8-bit integer values.


16-bit floating point value (FP16)
S E E E E E M M M M M M M M M M
1 bit
5 bit
10 bits
Quantization
8-bit integer value (INT8)
S M M M M M M M
1 bit
7 bits
S Sign
E Exponent
M Mantissa

Weight quantization from FP16 to INT8 | Image by author

(2) Tools and Data

The following diagram illustrates the architecture of the document knowledge Q&A application we will build in this project.



Document Q&A Architecture | Image by author

The file we will run the document Q&A on is the public 177-page 2022 annual report of Manchester United Football Club.

> Data Source: *Manchester United Plc (2022). 2022 Annual Report 2022 on Form 20-F. https://ir.manutd.com/~/media/Files/M/Manutd-IR/documents/manu-20f-2022-09-24.pdf (CC0: Public Domain, as SEC content is public domain and free to use)*

The local machine for this project has an **AMD Ryzen 5 5600X 6-Core Processor** coupled with **16GB RAM** (DDR4 3600). While it also has an RTX 3060TI GPU (8GB VRAM), it will not be used in this project since we will focus on CPU usage.

Let us now explore the software tools we will leverage in building this backend application:

(i) LangChain

LangChain is a popular framework for developing applications powered by language models. It provides an extensive set of integrations and data connectors, allowing us to chain and orchestrate different modules to create advanced use cases like chatbots, data analysis, and document Q&A.

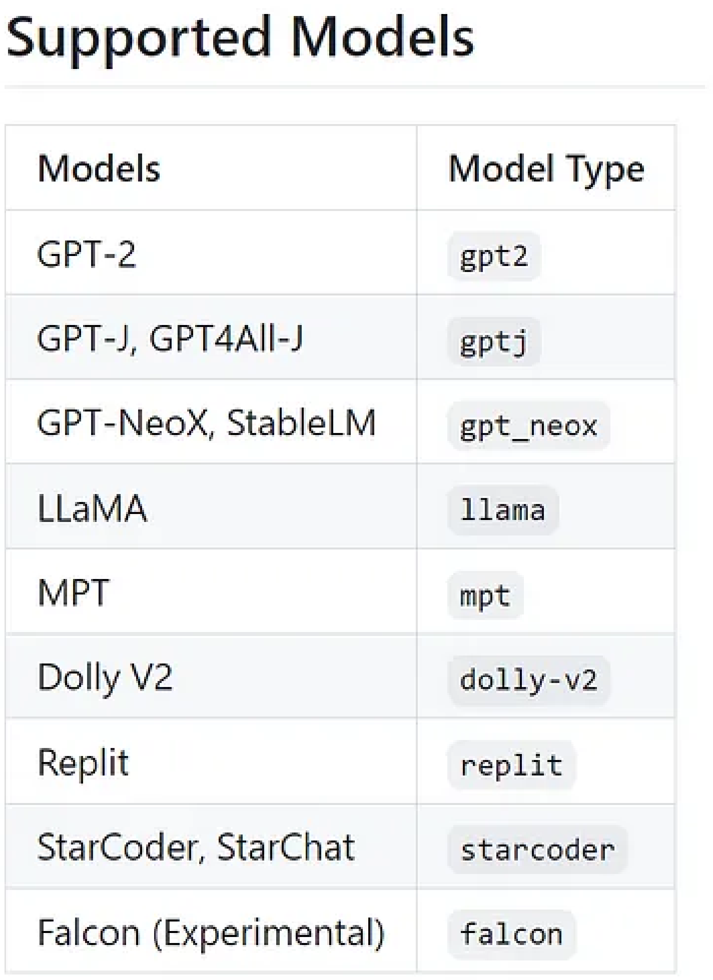
(ii) C Transformers

C Transformers is the Python library that provides bindings for transformer models implemented in C/C++ using the GGML library. At this point, let us first understand what GGML is about.

Built by the team at ggml.ai, the GGML library is a tensor library designed for machine learning, where it enables large models to be run on consumer hardware with high performance. This is achieved through integer quantization support and built-in optimization algorithms.

As a result, GGML versions of LLMs (quantized models in binary formats) can be run performantly on CPUs. Given that we are working with Python in this project, we will use the C Transformers library, which essentially offers the Python bindings for the GGML models.

C Transformers supports a selected set of open-source models, including popular ones like Llama, GPT4All-J, MPT, and Falcon.



Supported Models

Models	Model Type
GPT-2	`gpt2`
GPT-J, GPT4All-J	`gptj`
GPT-NeoX, StableLM	`gpt_neox`
LLaMA	`llama`
MPT	`mpt`
Dolly V2	`dolly-v2`
Replit	`replit`
StarCoder, StarChat	`starcoder`
Falcon (Experimental)	`falcon`

LLMs (and corresponding model type name) supported on C Transformers | Image by author

(iii) Sentence-Transformers Embeddings Model

sentence-transformers is a Python library that provides easy methods to compute embeddings (dense vector representations) for sentences, text, and images.

It enables users to compute embeddings for more than 100 languages, which can then be compared to find sentences with similar meanings.

We will use the open-source all-MiniLM-L6-v2 model for this project because it offers optimal speed and excellent general-purpose embedding quality.

(iv) FAISS

Facebook AI Similarity Search (FAISS) is a library designed for efficient similarity search and clustering of dense vectors.

Given a set of embeddings, we can use FAISS to index them and then leverage its powerful semantic search algorithms to search for the most similar vectors within the index.

Although it is not a full-fledged vector store in the traditional sense (like a database management system), it handles the storage of vectors in a way optimized for efficient nearest-neighbor searches.

(v) Poetry

Poetry is used for setting up the virtual environment and handling Python package management in this project because of its ease of use and consistency.

Having previously worked with venv, I highly recommend switching to Poetry as it makes dependency management more efficient and seamless.

Check out this video to get started with Poetry.

Join Medium with Kenneth's referral link

As a Medium member, a portion of your membership fee goes to writers you read, and you get full access to every story

kennethleungty.medium.com

(3) Open-Source LLM Selection

There has been tremendous progress in the open-source LLM space, and the many LLMs can be found on HuggingFace's Open LLM leaderboard.

I chose the latest open-source **Llama-2–7B-Chat model** (GGML 8-bit) for this project based on the following considerations:

Model Type (Llama 2)

- It is an open-source model supported in the C Transformers library.
- Currently the top performer across multiple metrics based on its Open LLM leaderboard rankings (as of July 2023).
- Demonstrates a huge improvement on the previous benchmark set by the original Llama model.
- It is widely mentioned and downloaded in the community.

Model Size (7B)

- Given that we are performing document Q&A, the LLM will primarily be used for the relatively simple task of summarizing document chunks. Therefore, the 7B model size fits our needs as we technically do not require an overly large model (e.g., 65B and above) for this task.

Fine-tuned Version (Llama-2-7B-Chat)

- The Llama-2-7B base model is built for text completion, so it lacks the fine-tuning required for optimal performance in document Q&A use cases.
- The Llama-2–7B-Chat model is the ideal candidate for our use case since it is designed for conversation and Q&A.
- The model is licensed (partially) for commercial use. It is because the fine-tuned model Llama-2-Chat model leverages publicly available instruction datasets and over 1 million human annotations.

Quantized Format (8-bit)

- Given that the RAM is constrained to 16GB, the 8-bit GGML version is suitable as it only requires a memory size of 9.6GB.
- The 8-bit format also offers a comparable response quality to 16-bit.
- The original unquantized 16-bit model requires a memory of ~15 GB, which is too close to the 16GB RAM limit.
- Other smaller quantized formats (i.e., 4-bit and 5–bit) are available, but they come at the expense of accuracy and response quality.

(4) Step-by-Step Guide

Now that we know the various components, let us go through the step-by-step guide on how to build the document Q&A application.

> The accompanying codes for this guide can be found in **this GitHub repo**, and all the dependencies can be found in the requirements.txt file.

Note: *Since many tutorials are already out there, we will* ***not*** *be deep diving into the intricacies and details of the general document Q&A components (e.g., text chunking, vector store setup). We will instead focus on the open-source LLM and CPU inference aspects in this article.*

Step 1 — Process data and build vector store

In this step, three sub-tasks will be performed:

- Data ingestion and splitting text into chunks
- Load embeddings model (sentence-transformers)
- Index chunks and store in FAISS vector store

```python
# File: db_build.py

from langchain.vectorstores import FAISS
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.document_loaders import PyPDFLoader, DirectoryLoader
from langchain.embeddings import HuggingFaceEmbeddings

# Load PDF file from data path
loader = DirectoryLoader('data/',
                         glob="*.pdf",
                         loader_cls=PyPDFLoader)
documents = loader.load()

# Split text from PDF into chunks
text_splitter = RecursiveCharacterTextSplitter(chunk_size=500,
                                               chunk_overlap=50)
texts = text_splitter.split_documents(documents)

# Load embeddings model
embeddings = HuggingFaceEmbeddings(model_name='sentence-transformers/all-MiniLM-L6-v2',
                                   model_kwargs={'device': 'cpu'})

# Build and persist FAISS vector store
vectorstore = FAISS.from_documents(texts, embeddings)
vectorstore.save_local('vectorstore/db_faiss')
```

db_build.py hosted with ❤ by GitHub view raw

After running the Python script above, the vector store will be generated and saved in the local directory named `'vectorstore/db_faiss'`, and is ready for semantic search and retrieval.

Step 2 — Set up prompt template

Given that we use the Llama-2–7B-Chat model, we must be mindful of the prompt templates utilized here.

For example, OpenAI's GPT models are designed to be conversation-in and message-out. It means input templates are expected to be in a chat-like transcript format (e.g., separate system and user messages).

However, those templates would not work here because our Llama 2 model is not specifically optimized for that kind of conversational interface. Instead, a classic prompt template like the one below would be preferred.

```python
# File: prompts.py

qa_template = """Use the following pieces of information to answer the user's question.
If you don't know the answer, just say that you don't know, don't try to make up an answer.

Context: {context}
Question: {question}

Only return the helpful answer below and nothing else.
Helpful answer:
"""
```

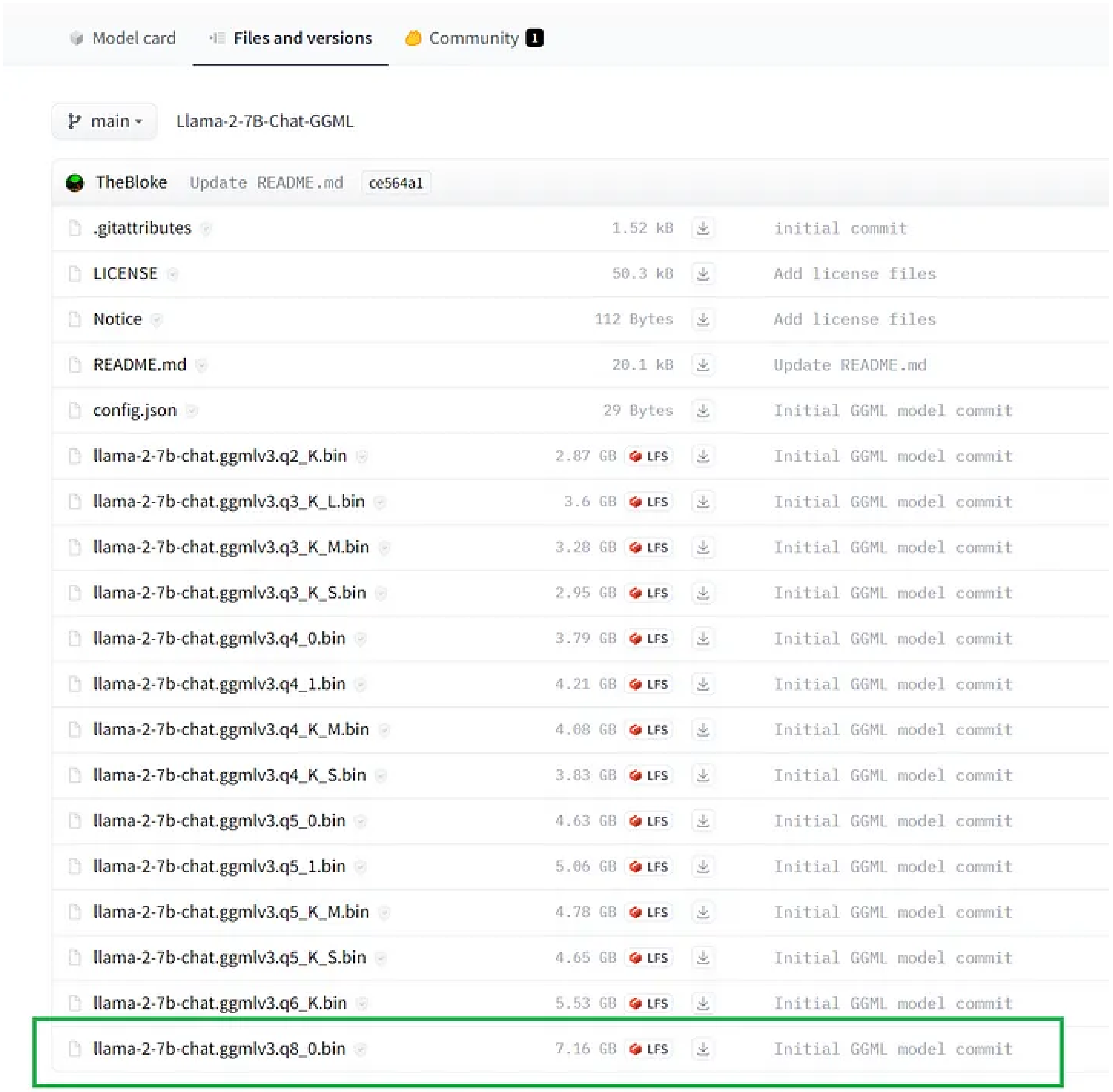
prompts.py hosted with ❤ by GitHub view raw

Note: The relatively smaller LLMs, like the 7B model, appear particularly sensitive to formatting. For instance, I got slightly different outputs when I altered the whitespaces and indentation of the prompt template.

Step 3 — Download the Llama-2–7B-Chat GGML binary file

Since we will be running the LLM locally, we need to download the binary file of the quantized Llama-2–7B-Chat model.

We can do so by visiting TheBloke's Llama-2–7B-Chat GGML page hosted on Hugging Face and then downloading the GGML 8-bit quantized file named `llama-2-7b-chat.ggmlv3.q8_0.bin`.



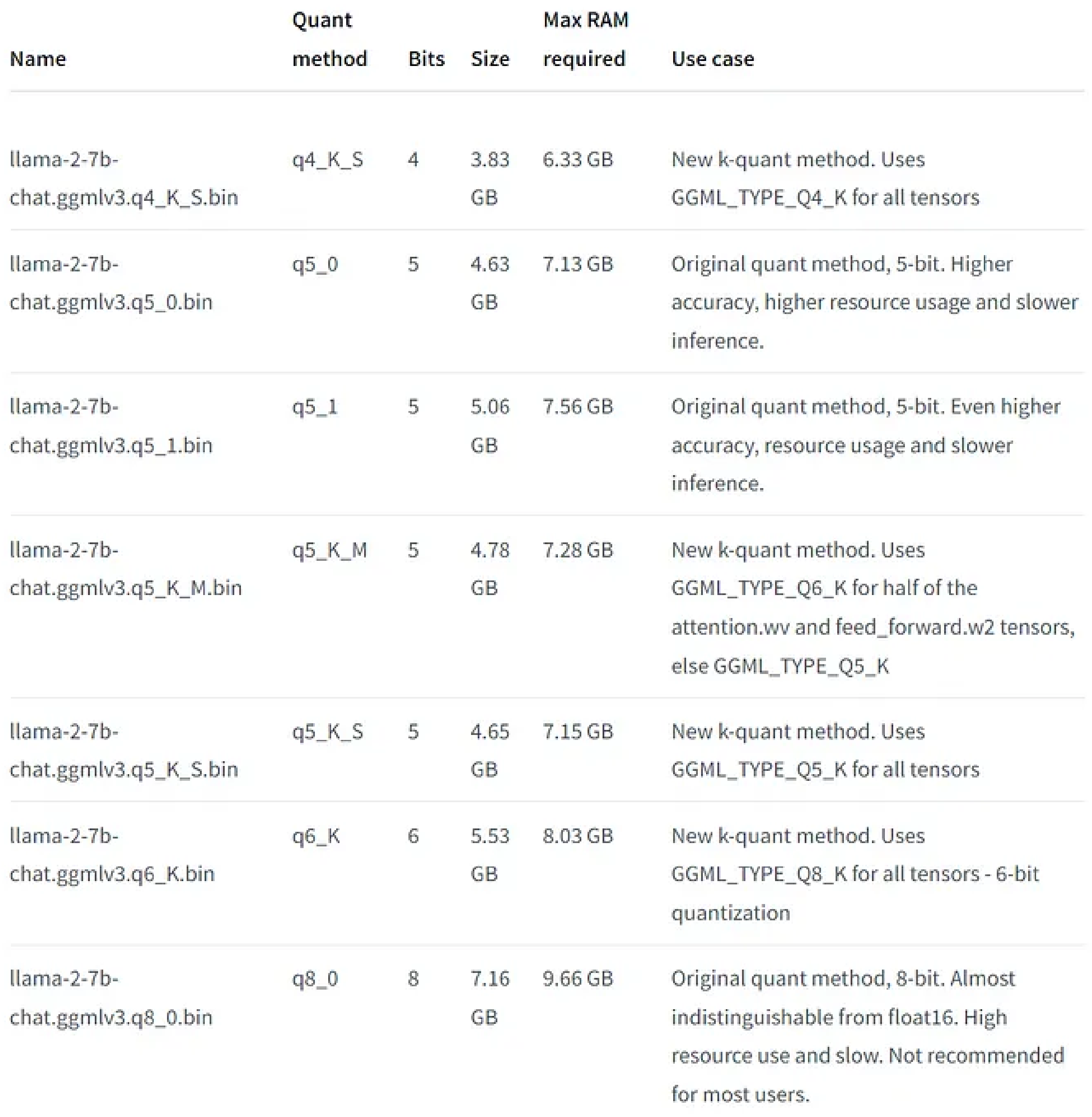
TheBloke Update README.md ce564a1

.gitattributes	1.52 kB		initial commit
LICENSE	50.3 kB		Add license files
Notice	112 Bytes		Add license files
README.md	20.1 kB		Update README.md
config.json	29 Bytes		Initial GGML model commit
llama-2-7b-chat.ggmlv3.q2_K.bin	2.87 GB	LFS	Initial GGML model commit
llama-2-7b-chat.ggmlv3.q3_K_L.bin	3.6 GB	LFS	Initial GGML model commit
llama-2-7b-chat.ggmlv3.q3_K_M.bin	3.28 GB	LFS	Initial GGML model commit
llama-2-7b-chat.ggmlv3.q3_K_S.bin	2.95 GB	LFS	Initial GGML model commit
llama-2-7b-chat.ggmlv3.q4_0.bin	3.79 GB	LFS	Initial GGML model commit
llama-2-7b-chat.ggmlv3.q4_1.bin	4.21 GB	LFS	Initial GGML model commit
llama-2-7b-chat.ggmlv3.q4_K_M.bin	4.08 GB	LFS	Initial GGML model commit
llama-2-7b-chat.ggmlv3.q4_K_S.bin	3.83 GB	LFS	Initial GGML model commit
llama-2-7b-chat.ggmlv3.q5_0.bin	4.63 GB	LFS	Initial GGML model commit
llama-2-7b-chat.ggmlv3.q5_1.bin	5.06 GB	LFS	Initial GGML model commit
llama-2-7b-chat.ggmlv3.q5_K_M.bin	4.78 GB	LFS	Initial GGML model commit
llama-2-7b-chat.ggmlv3.q5_K_S.bin	4.65 GB	LFS	Initial GGML model commit
llama-2-7b-chat.ggmlv3.q6_K.bin	5.53 GB	LFS	Initial GGML model commit
llama-2-7b-chat.ggmlv3.q8_0.bin	7.16 GB	LFS	Initial GGML model commit

Files and versions page of Llama-2–7B-Chat-GGML page on HuggingFace | Image by author

The downloaded `.bin` file for the 8-bit quantized model can be saved in a suitable project subfolder like `/models`.

The model card page also displays more information and details for each quantized format:



Name	Quant method	Bits	Size	Max RAM required	Use case
llama-2-7b-chat.ggmlv3.q4_K_S.bin	q4_K_S	4	3.83 GB	6.33 GB	New k-quant method. Uses GGML_TYPE_Q4_K for all tensors
llama-2-7b-chat.ggmlv3.q5_0.bin	q5_0	5	4.63 GB	7.13 GB	Original quant method, 5-bit. Higher accuracy, higher resource usage and slower inference.
llama-2-7b-chat.ggmlv3.q5_1.bin	q5_1	5	5.06 GB	7.56 GB	Original quant method, 5-bit. Even higher accuracy, resource usage and slower inference.
llama-2-7b-chat.ggmlv3.q5_K_M.bin	q5_K_M	5	4.78 GB	7.28 GB	New k-quant method. Uses GGML_TYPE_Q6_K for half of the attention.wv and feed_forward.w2 tensors, else GGML_TYPE_Q5_K
llama-2-7b-chat.ggmlv3.q5_K_S.bin	q5_K_S	5	4.65 GB	7.15 GB	New k-quant method. Uses GGML_TYPE_Q5_K for all tensors
llama-2-7b-chat.ggmlv3.q6_K.bin	q6_K	6	5.53 GB	8.03 GB	New k-quant method. Uses GGML_TYPE_Q8_K for all tensors - 6-bit quantization
llama-2-7b-chat.ggmlv3.q8_0.bin	q8_0	8	7.16 GB	9.66 GB	Original quant method, 8-bit. Almost indistinguishable from float16. High resource use and slow. Not recommended for most users.

Different quantized formats with details | Image by author

Note: *To download other GGML quantized models supported by C Transformers, visit the main TheBloke page on HuggingFace to search for your desired model and look for the links with names that end with '-GGML'.*

Step 4— Setup LLM

To utilize the GGML model we downloaded, we will leverage the integration between C Transformers and LangChain. Specifically, we will use the CTransformers LLM wrapper in LangChain, which provides a unified interface for the GGML models.

```python
# File: llm.py
from langchain.llms import CTransformers

# Local CTransformers wrapper for Llama-2-7B-Chat
llm = CTransformers(model='models/llama-2-7b-chat.ggmlv3.q8_0.bin', # Location of downloaded GGML model
                    model_type='llama', # Model type Llama
                    config={'max_new_tokens': 256,
                            'temperature': 0.01})
```

llm.py hosted with ❤ by GitHub view raw

We can define a host of configuration settings for the LLM, such as maximum new tokens, top k value, temperature, and repetition penalty.

Note: I set the temperature as 0.01 instead of 0 because I got odd responses (e.g., a long repeated string of the letter E) when the temperature was exactly zero.

Step 5 — Build and initialize RetrievalQA

With the prompt template and C Transformers LLM ready, we write three functions to build the LangChain RetrievalQA object that enables us to perform document Q&A.

```python
# File: utils.py
from langchain import PromptTemplate
from langchain.chains import RetrievalQA
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.vectorstores import FAISS

# Wrap prompt template in a PromptTemplate object
def set_qa_prompt():
    prompt = PromptTemplate(template=qa_template,
                            input_variables=['context', 'question'])
    return prompt


# Build RetrievalQA object
def build_retrieval_qa(llm, prompt, vectordb):
    dbqa = RetrievalQA.from_chain_type(llm=llm,
                                       chain_type='stuff',
                                       retriever=vectordb.as_retriever(search_kwargs={'k':2}),
                                       return_source_documents=True,
                                       chain_type_kwargs={'prompt': prompt})
    return dbqa


# Instantiate QA object
def setup_dbqa():
    embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2",
                                       model_kwargs={'device': 'cpu'})
    vectordb = FAISS.load_local('vectorstore/db_faiss', embeddings)
    qa_prompt = set_qa_prompt()
    dbqa = build_retrieval_qa(llm, qa_prompt, vectordb)

    return dbqa
```

utils.py hosted with ❤ by GitHub view raw

Step 6 — Combining into the main script

The next step is to combine the previous components into the `main.py` script. The `argparse` module is used because we will pass our user query into the application from the command line.

Given that we will return source documents, additional code is appended to process the document chunks for a better visual display.

To evaluate the speed of CPU inference, the `timeit` module is also utilized.

```python
# File: main.py
import argparse
import timeit

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('input', type=str)
    args = parser.parse_args()
    start = timeit.default_timer() # Start timer

    # Setup QA object
    dbqa = setup_dbqa()

    # Parse input from argparse into QA object
    response = dbqa({'query': args.input})
    end = timeit.default_timer() # End timer

    # Print document QA response
    print(f'\nAnswer: {response["result"]}')
    print('='*50) # Formatting separator

    # Process source documents for better display
    source_docs = response['source_documents']
    for i, doc in enumerate(source_docs):
        print(f'\nSource Document {i+1}\n')
        print(f'Source Text: {doc.page_content}')
        print(f'Document Name: {doc.metadata["source"]}')
        print(f'Page Number: {doc.metadata["page"]}\n')
        print('='* 50) # Formatting separator

    # Display time taken for CPU inference
    print(f"Time to retrieve response: {end - start}")
```

main.py hosted with ❤ by GitHub view raw

Step 7 — Running a sample query

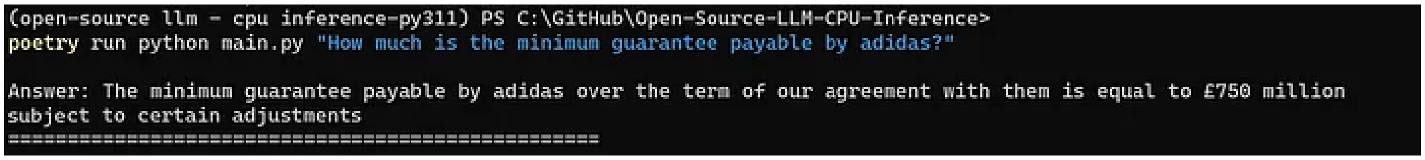
It is now time to put our application to the test. Upon loading the virtual environment from the project directory, we can run a command in the command line interface (CLI) that comprises our user query.

For example, we can ask about the value of the minimum guarantee payable by Adidas (Manchester United's global technical sponsor) with the following command:

```
poetry run python main.py "How much is the minimum guarantee payable by adidas?"
```

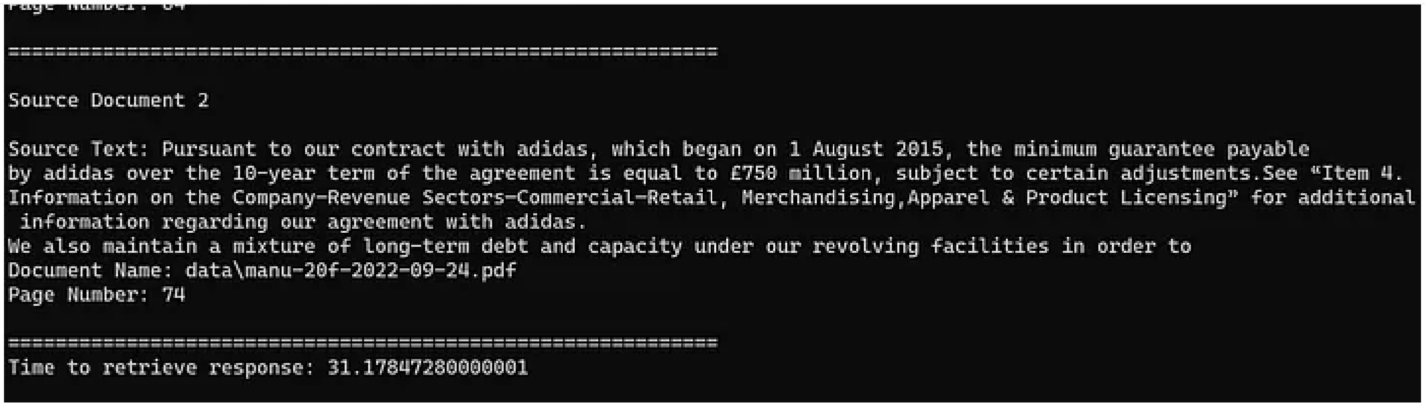
Note: If we are not using Poetry, we can omit the prepended `poetry run`.

Results



```
(open-source llm - cpu inference-py311) PS C:\GitHub\Open-Source-LLM-CPU-Inference>
poetry run python main.py "How much is the minimum guarantee payable by adidas?"

Answer: The minimum guarantee payable by adidas over the term of our agreement with them is equal to £750 million
subject to certain adjustments
==================================================
```



```
Page Number: 64

============================================================

Source Document 2

Source Text: Pursuant to our contract with adidas, which began on 1 August 2015, the minimum guarantee payable
by adidas over the 10-year term of the agreement is equal to £750 million, subject to certain adjustments.See "Item 4.
Information on the Company-Revenue Sectors-Commercial-Retail, Merchandising,Apparel & Product Licensing" for additional
 information regarding our agreement with adidas.
We also maintain a mixture of long-term debt and capacity under our revolving facilities in order to
Document Name: data\manu-20f-2022-09-24.pdf
Page Number: 74

============================================================
Time to retrieve response: 31.17847280000001
```

Output from user query passed into document Q&A application | Image by author

The output shows that we successfully obtained the correct response for our user query (i.e., £750 million), along with the relevant document chunks that are semantically similar to the query.

The total time of 31 seconds for launching the application and generating a response is pretty good, given that we are running it locally on an AMD Ryzen 5600X (which is a good CPU but by no means the best in the market currently).

The result is even more impressive given that running LLM inference on GPUs (e.g., directly on HuggingFace) can also take double-digit seconds.

Your Mileage May Vary

Depending on your CPU, the time taken to obtain a response may vary. For example, when I test it out on my laptop, it could go into the range of several minutes.

The thing to note is that getting LLMs to fit into consumer hardware is still in the early stages, so we cannot expect speeds that are on par with OpenAI APIs (which are driven by loads of computing power).

For now, one can certainly consider running this on a more powerful CPU instance, or switching to using GPU instances (such as free ones on Google Colab).

(5) Next Steps

Now that we have built a document Q&A backend LLM application that runs on CPU inference, there are many exciting steps we can take to bring this project forward.

- Build a frontend chat interface with Streamlit, especially since it has made two major announcements recently: Integration of Streamlit with LangChain, and the launch of Streamlit ChatUI to build powerful chatbot interfaces easily.
- Dockerize and deploy the application on a cloud CPU instance. While we have explored local inference, the application can easily be ported to the cloud. We can also leverage more powerful CPU instances on the cloud to speed up inference (e.g., compute-optimized AWS EC2 instances like c5.4xlarge)
- Experiment with slightly larger LLMs like the Llama 13B Chat model. Since we have worked with 7B models, assessing the performance of slightly larger ones is a good idea since they should theoretically be more accurate and still fit within memory.

- Experiment with smaller quantized formats like the 4-bit and 5-bit (including those with the new k-quant method) to objectively evaluate the differences in inference speed and response quality.
- Leverage local GPU to speed up inference. If we want to test the use of GPUs on the C Transformers models, we can do so by running some of the model layers on the GPU. It is useful because Llama is the only model type that has GPU support currently.
- Evaluate the use of vLLM, a high-throughput and memory-efficient inference and serving engine for LLMs. However, utilizing vLLM requires the use of GPUs.

I will work on articles and projects addressing the above ideas in the upcoming weeks, so stay tuned for more insightful generative AI content!

Before you go

I welcome you to **join me on a data science learning journey!** Follow this Medium page and check out my GitHub to stay in the loop of more exciting practical data science content. Meanwhile, have fun running open-source LLMs on CPU inference!

arXiv Keyword Extraction and Analysis Pipeline with KeyBERT and Taipy
Build a keyword analysis application in Python comprising a frontend user interface and backend pipeline
towardsdatascience.com

How to Dockerize Machine Learning Applications Built with H2O, MLflow, FastAPI, and Streamlit
An easy-to-follow guide to containerizing multi-service ML applications with Docker
towardsdatascience.com

Micro, Macro & Weighted Averages of F1 Score, Clearly Explained
Understanding the concepts behind the micro average, macro average and weighted average of F1 score in multi-class classification.
towardsdatascience.com

Large Language Models | Langchain | Machine Learning | Python | Hands On Tutorials

Follow

Written by Kenneth Leung

7K Followers · Writer for Towards Data Science

Data Scientist at Boston Consulting Group (BCG) | Tech Writer | 1.3M+ views on Medium | linkedin.com/in/kennethleungty | github.com/kennethleungty