Ioannis Chalkiadakis*

# Dataset Recommendation with Large Language Models

**Abstract:** Notes to set up computing environment in Linux for the PIR "Dataset Recommendation with LLM" project at INSA Lyon.
Supervisors: Dr Malcolm Egan (INRIA), Dr Ioannis Chalkiadakis (CNRS).

## 1 Virtual environment setup

Starting code has been placed at the following GitHub repository: https://github.com/ichalkiad/dataLLMrec. Please clone (or download) the repository and set up a virtual environment to facilitate installation of software. For example, using Anaconda, run in the command line:

```
    conda env create -f datarec.yml
```

to create the "pir_env" environment and then activate it by running

```
    conda activate pir_env
```

## 2 LLM inference framework - llama.cpp

In order to experiment with Large Language Models capable enough, we need to install an efficient inference engine that will allow us to run a big LLM locally on a personal computer without access to specialised hardware (e.g. GPU, TPU).

For this purpose, we will utilise llama.cpp, a highly optimised inference engine for running quantised LLM models, which works well on CPU-only systems. In LLM literature, quantisation is a technique that reduces the precision of the floating point numbers used to represent a model's weights. LLMs are typically trained using 32-bit floating-point (FP32) or 16-bit floating-point (FP16) precision. These high-precision weights are converted to lower precision formats like 8-bit integers (INT8), 4-bit integers (INT4), or even lower (1-bit[1]), which significantly reduces memory requirements and computational load while attempting to preserve model performance.

Briefly, a simple quantisation scheme works as follows:

1. The distribution of weights in the model is analysed;
2. Appropriate scaling factors are calculated to map the range of floating-point values to the smaller integer range;
3. Each weight is converted from floating-point to integer format using these scaling factors;
4. The quantised weights plus scaling factors are stored, requiring much less memory.

The trade-off is between model size/speed and accuracy. More aggressive quantization (lower bit-width) means smaller size but potentially more quality degradation.

There are more advanced quantisation methods, which are used in efficient file formats for storing and distributing LLMs. One such format is GGUF (GPT-Generated Unified Format), which is designed

**1** https://github.com/microsoft/BitNet specifically for efficiently storing and accessing quantised language models. A GGUF file includes model weights, architecture information, and metadata in one file, yet it allows loading only the parts of the model needed at runtime.

The llama.cpp framework[2] is a software framework written in C/C++ which will allow us to load models stored in GGUF format and run them on non-specialised laptops. First we will install llama.cpp and then its Python API to be able to call llama.cpp methods via a Python program.

In your command line, run the following sequence of steps.

1. Install system-wide pre-requisites and download llama.cpp repository

   –
   ```
   sudo apt update
   sudo apt install build-essential cmake
   sudo apt install ccache
   sudo apt install libcurl4-openssl-dev
   ```

   ```
   git clone https://github.com/ggerganov/llama.cpp
   ```

   –
   ```
   mkdir build && cd build
   cmake .. && cmake --build . --config Release
   ```

2. Install llama-cpp-python package, which provides Python bindings for llama.cpp
   – To use system's OpenMP multiprocessing API run the following:

     ```
     export CXX=/usr/bin/g++
     export CC=/usr/bin/gcc
     ```

     Alternatively, the virtual environment's OpenMP installation may be used but ensure it is appropriately linked.
   – Ensure it can access efficient vector processing extensions to the instruction set architecture for Intel/AMD processors

     ```
     grep -q avx2 /proc/cpuinfo && echo "AVX2 Supported" || echo "AVX2 Not Supported"
     ```

   – If AVX2 supported:

     ```
     CMAKE_ARGS="-DGGML_BLAS=ON -DLLAMA_AVX2=ON -DGGML_BLAS_VENDOR=OpenBLAS" pip
         install llama-cpp-python
     ```

   – Otherwise run:

     ```
     CMAKE_ARGS="-DGGML_BLAS=ON -DLLAMA_AVX=ON -DGGML_BLAS_VENDOR=OpenBLAS" pip
         install llama-cpp-python
     ```

## 3 LLM download

Finally, we will need to download a Large Language Model to play around with. One such option is the Phi-3-mini model originally developed by Microsoft[3]:

– Parameter count: 3.8 billion parameters
– Context length: 4K tokens
– Good reasoning capabilities for its size

**2** https://github.com/ggml-org/llama.cpp

**3** https://news.microsoft.com/source/features/ai/the-phi-3-small-language-models-with-big-potential/
https://azure.microsoft.com/en-us/blog/introducing-phi-3-redefining-whats-possible-with-slms/
https://export.arxiv.org/abs/2404.14219

- Optimized for CPU inference
- Inference speed: 5 – 15 tokens per second on a good laptop CPU
- Memory usage: 6 – 8 GB RAM
- Reasonable for assistant-style interactions and text generation

In the main repository folder, download the following model hosted on HuggingFace:

https://huggingface.co/SanctumAI/Phi-3-mini-4k-instruct-GGUF/resolve/main/phi-3-mini-4k-instruct.Q4_K_M.gguf

## 4 Repository code

The source file 'dataset_recommendation_llamacpp.py' provides a starting point to play with Phi-3 in the context of our tast. You can experiment with prompting strategies (temperature, top-K, top-p) via the command line, e.g.:

```
python datarec/src/dataset_recommendation_llamacpp.py --top_k=5 --top_p=0.5 --temp=0.3
```

or you can edit the file to modify the prompt (more practical than passing it via the command line).

A summary of the aforementioned prompting strategies is:[4]

- Temperature: a parameter that controls the degree of randomness in token selection. Lower temperatures are good for prompts that are likely to have a more deterministic response, while higher temperatures can produce more creative or more diverse. Setting the temperature to 0 (greedy decoding) is deterministic in the next token selection: the highest probability token is always selected. If more than one token have the same probability, then the tie may be broken at random or some other method, e.g top-K or top-p. Temperatures close to the max (1) tend to create more random output, and as temperature increases all tokens become equally likely to be the next predicted token.
- Top-K sampling: selects the top K most likely tokens from the model's predicted distribution.
- Top-p sampling selects the top tokens whose cumulative probability does not exceed a certain value (p)
- If temperature, top-K, and top-P are all set, tokens that meet both the top-K and top-p criteria are candidates for the next predicted token, and then temperature is applied to sample from the tokens that passed the top-K and top-p criteria. If temperature is not set, whatever tokens meet the top-K and/or top-p criteria are then randomly selected from to produce a single next predicted token.

The script will query the LLM for possible tasks given a dataset name, author and description and the output will be in the following format:

```
"Task 1. This dataset is good for text generation problems that involve mathematical
    expressions.\n"
"Task 2. The data can be used to improve the reasoning capabilities of large language
    models."
```

It will then parse the output to feed each task back to the LLM, extract the embeddings per token (according to the LLM tokenization), average the token embeddings and save the output in a text file, where each line is in JSON format with the following structure:

```
{"dataset": dataset-name, "dataset_author": dataset-author, "dataset_embedding":
    dataset_embedding_vec_avg, "avg_task_i_embedding": total-task-embedding-vector}.
```

You can then read the file to extract the embedding vector of the dataset or the embedding vector per task and proceed with your research into assessing their similarity.

4 If interested, more details on prompt engineering can be found here : https://www.kaggle.com/whitepaper-prompt-engineering.