Training a Language Model to Book Flights and/or Hotel Rooms

Denis Kazakov

July 2025

Abstract

This is a test task to show model training skills.

https://github.com/Denis-Kazakov/hotel_booking

1. Assumptions and simplifications

The key assumption is that the model is supposed to send requests to a single hotel booking service with a documented API with two endpoints: one for getting information on available hotels and the other for booking a room. For the first endpoint, the user is supposed to provide the city of interest as well as both check-in and check-out dates; for the second, the city, name of the hotel and check-in and check-out dates.

Code in main.py simulates the booking service, which checks a fake database in hotel_db.csv. The service returns requested data or a booking confirmation and there are available hotels in the database. If some of the required data are missing, the service returns an error message.

The LLM is supposed to perform the basic booking functions: receive user queries and check for missing information. If some data are missing, it should explain the user what is needed. If the user’s question is not related to hotel booking, it should respond with a description of its tasks. If all data are provided, the LLM should send a request to the booking API and then pass the outcome to the user: data on available hotels, booking confirmation or information that there are no available hotels or rooms on given dates. Importantly, the model should only return data from the API and should not invent information.

Limitations

The test required using an open-source model with no more than 7B parameters. The work was done on a computer with 8 GB vRAM, so the models had to be quantized to 4 bits which may have impacted on the results. This also necessitated low-rank adaptation and other measures such as gradient accumulation.

2. Solution architecture

User queries are processed in two stages with two calls to the LLM each with its own system prompt and other parameters: first, the LLM examines the request for missing data and, if everything is OK, generates a tool call sending a request to the booking API. At the second stage, when the API response or error message is received, the LLM uses it with the original user query to generate final response to the user.

Given the hardware limitations, I used model Mistral 7B, version 0.3, in Ollama, which allegedly supported tool calling.

3. Baseline

The baseline case was a ready model that had not been fine-tuned for the task, except maybe few-shot learning. Some prompt-engineering helped to design baseline code given in 01_booking_baseline.ipynb, which seemed to work most of the time. It was discovered when the study was already underway that tool calling in Ollama did not work as described so I had to use structured output as a workaround.

Dataset

Same model was used to generate a dataset, which included both valid queries containing all required data and invalid queries with some data missing or unrelated to hotel booking. Some of the cities and hotels requested in the dataset queries were available, some were not to test the model's ability to handle both successful and unsuccessful results. Code fo dataset generation is given in 02_generate_dataset.ipynb and 03_train_test_split.ipynb.

The full dataset was saved in dataset_queries_only.xlsx. Column *available* indicates whether the requested information or hotel will be available from the API or not, serving as an extra safeguard to detect hallucinations.

Dataset size in this case is limited by manual effort required to verify model outputs.

Evaluation

The baseline code was run on all dataset queries (one change was to save both 1st and 2nd stage responses) as shown in 04_test_baseline.ipynb. The results were manually evaluated and saved in baseline_test_results.xlsx.

Percentages of correct answers were calculated in 05_baseline_eval.ipynb at 79% for the 1st stage and 60% for the 2nd stage. One reason for low second-stage marks is that the model often erroneously believed the requested hotel to be in Moscow, probably taking this from the system prompt.

4. Fine-tuning

Since there two different calls to the LLM are made for each user query, the plan was to first fine-tune for the 1st stage, evaluate the results and then fine-tune for the 2nd stage.

Dataset

The baseline code was run on the train set and then the resulting raw datset was manually reviewed corrected to create ground truth labels for the 1st stage. The ground truth labels in this case are stringified JSONs with function calls supplemented with the system prompt and tool descriptions. (07_prepare_trainset.ipynb)

Model

Ollama does not support fine-tuning, so the same model (Mistral-7B-Instruct-v0.3) was downloaded from Hugging Face and quantized with bitsandbytes to 4 bits (06_quantize_model.ipynb)

Fine-tuning

As the train set was not very large, the model was fine-tuned with LoRA on the whole train set for one training epoch to prevent overfitting (08_finetune_1st_stage.ipynb). A larger dataset would allow splitting a holdout set that would be used to track training progress and stop it at the lowest validation loss.

The fine-tuned LoRA adapter was merged with the downloaded unquatized model, which was then converted to GGUF with llama.cpp and converted to the Ollama format with 4-bit quantization (09_convert_to_ollama.ipynb).

Evaluation

The same test set was run through the fine-tuned model (10_test_1st_stage_finetune.ipynb) and the percentages of correct outputs were calculated again (11_eval_1st_stage_finetune.ipynb).

Unfortunately, the 1st stage results did not change (the 2nd stage results improved by 6% but this cannot be attributed to fine-tuning).

5. Discussion of results

Fine-tuning did not improve the model’s performance.

Possible next steps:

1. Use other platforms than Ollama with better support for tool calling, ideally tool calling and fine-tuning within the same framework.

2. Generate a much larger dataset. Firs-stage output is function calls conforming to a certain schema so it can be generated with code.

3. Use a holdout set to optimize training hyperparameters.