Fine-tuning GPT-3.5-Turbo for Natural Language to SQL

Mo Pourreza · Follow
Published in Dataherald
10 min read · 4 hours ago

Listen Share

Photo by Mariia Shalabaieva on Unsplash

Background

Allowing non-technical users to ask questions from a database has been a problem of interest in academia and industry for years. The recent advances in Large Language Model (LLM) technology, such as GPT-4, have improved the accuracy of proposed solutions. However, since the most advanced LLMs have not been open for fine-tuning, recent work in the space has focused on creating Retrieval-Augmented Generation (RAG) algorithms that can enable complex Natural Language to SQL (NL-to-SQL) scenarios without modifying the underlying LLM.

Last week, OpenAI opened up GPT-3.5-turbo for fine-tuning. In this post, we will fine-tune our own NL-to-SQL model and compare its performance against the state of the art RAG approach. We will use the Spider dataset from Yale university as our test benchmark.

Fine-tuning GPT-3.5-Turbo for NL-to-SQL

Like all model training and fine-tuning, the first step of fine-tuning GPT-3.5-Turbo is the creation and upload of a training dataset. Since GPT-3.5-Turbo is a ChatModel, this dataset must use to the following format, and be uploaded as a JSONL file

```
{"messages": [{"role": "system", "content": "system_prompt"}, {"role": "user", "content": "user_
{"messages": [{"role": "system", "content": "system_prompt"}, {"role": "user", "content": "user_
{"messages": [{"role": "system", "content": "system_prompt"}, {"role": "user", "content": "user_
```

The Spider dataset has a holdout test set of 2147 question/SQL pairs, a development set of 1034 question/SQL pairs, and a training set of 7000 question/SQL pairs. We will build our fine-tuning dataset in the structure above from the Spider training set.

Creating the training dataset

An NL-to-SQL task is defined as follows: **given a question and database, identify a SQL query that when executed against the database returns a result set that can answer the question.** Various approaches have been explored on how best to prompt LLMs for this task, and it is generally agreed that the prompt needs to include an instructional component, details of the database schema, information about the database's content, a set of task-specific demonstrations and of course the actual question at hand.

Given the format of the ChatModel training data, the elements above have to be presented within the following three prompts:

- **system_prompt** — will contain the instruction, database schema and database content
- **user_prompt** — will contain the natural language question
- **assistant_prompt** — where the SQL will be provided together with a reasoning step

Let's look at how to create each of these for our NL-to-SQL training dataset.

The system prompt

Creating the system_prompt is by far the most complex part of this exercise. At a minimum, the system_prompt needs to include:

1. The system instruction
2. The DB schema
3. Information about the DB content

In addition, for any real-world use case with a large number of tables, the samples in the training set should also train the model to select the correct tables from the DB for the SQL query (i.e perform schema-linking).

System Instruction

For the instruction we used the following standard prompt

```
You are an assistant that is an expert in generating Sqlite SQL queries.
Having the access to database content, generate a correct Sqlite SQL query for the given questic
### Database content ###
```

Database Schema

In the literature there are many proposed prompt formats for the database schema and content with no clear consensus around which performs best. We found the following to be the optimal representation of the database schema:

```
CREATE TABLE concert (
"concert_ID" INTEGER NOT NULL,
"concert_Name" TEXT NOT NULL, – the name of the concert
"Theme" TEXT, – theme of the concert
"Stadium_ID" TEXT NOT NULL,
"Year" TEXT, PRIMARY KEY ("concert_ID"),
FOREIGN KEY("Stadium_ID")
REFERENCES stadium ("Stadium_ID")
)
CREATE TABLE singer (
"Singer_ID" INTEGER NOT NULL,
"Name" TEXT, – name of the singer
```

```
    "Country" TEXT NOT NULL, - country where the singer born
    "Song_Name" TEXT NOT NULL, - the name of the song produced by the singer
    "Song_release_year" TEXT, - The release year of the song
    "Age" INTEGER,
    "Is_male" BOOLEAN NOT NULL,
    PRIMARY KEY ("Singer_ID")
    )
```

Database Content

After much experimentation we found the following template to perform the best at training the model about the database content:

```
/*
Columns in concert and 3 examples in each column for the high cardinality columns :
concert_ID: 1025 , 1101 , 1247
concert_Name : "Fire", "Dance", "Sky"
Stadium_ID : 9, 10, 11
*/
/*
Columns in concert and all categories for the low cardinality columns :
Theme : " ROCK ", " POP ", " HIP-HOP "
Year : 2022, 2021, 2023, 2020
*/
/*
Columns in concert and 3 examples in each column for the high cardinality columns :
Singer_ID : 10235 , 110231 , 1242447
Name : "Jordan", "Gabriel", "Tiffany"
Country : "Iran", "India", "Canada"
Song_Name : "dance in the fire", "rain", "sky"
Age : 19, 20, 21
*/
/*
Columns in concert and all categories for the low cardinality columns :
Is_male : "MALE", "FEMALE",
Song_release_year : 2022, 2021, 2023, 2020
*/
```

An important element in the database content is how to identify categorical (low cardinality) columns. The threshold for distinguishing between low and high cardinality columns depends on the context window size of the Large Language Model (LLM) being fine-tuned. Given the 4096 token context window of GPT-3.5-turbo, we determined 20 tokens as the appropriate threshold between low and high cardinality columns.

Schema Linking

The final challenge in creating the system_prompts for our training set is to provide samples in such a way that train the model to correctly perform schema-linking on the database. To do this, we employed the following heuristic: for each individual NL <> SQL sample we included a random selection of other tables from the DB in addition to the correct tables until we reached the context window limit of 4000 tokens. To mitigate the influence of positional information, we further randomized the order of tables. In short, each system_prompt included the schema and content of the relevant tables mixed in with other irrelevant tables, helping train the model in picking the correct tables for the query.

We will now put all of this together to build our system_prompts.

For the sample below from Spider:

```
Question : "How many heads of the departments are older than 56 ?"
SQL: "SELECT count(*) FROM head WHERE age > 56"
```

The system_prompt will be

```
You are an assistant that is an expert in generating Sqlite SQL queries.
Having the access to database content, generate a correct Sqlite SQL query for the given questic
### Database content ###
CREATE TABLE trip (
id INTEGER, duration INTEGER,
start_date TEXT,
start_station_name TEXT,
start_station_id INTEGER,
end_date TEXT,
end_station_name TEXT,
end_station_id INTEGER,
bike_id INTEGER,
subscription_type TEXT,
zip_code INTEGER,
PRIMARY KEY (id)
)
/* Columns in trip and 3 examples in each column for high cardinality columns :
id : 900645, 900752, 900524
duration : 1131, 2146, 1155
start_date : 8/21/2015 17:39, 8/21/2015 17:03, 8/21/2015 17:16
start_station_name : Howard at 2nd, 2nd at Folsom, Market at 10th
start_station_id : 56, 65, 49 end_date : 8/21/2015 17:19, 8/21/2015 18:08, 8/21/2015 17:32
```

```
end_station_name : Howard at 2nd, 2nd at Folsom, Market at 10th
end_station_id : 56, 65, 49
bike_id : 586, 56, 65
zip_code : 94070, 94530, 94040-1724
*/
/* Columns in trip and all categories for low cardinality columns :
subscription_type : Customer, Subscriber
*/

CREATE TABLE management (
"department_ID" INTEGER,
"head_ID" INTEGER,
temporary_acting TEXT,
PRIMARY KEY ("department_ID", "head_ID"),
FOREIGN KEY("head_ID") REFERENCES head ("head_ID"),
FOREIGN KEY("department_ID") REFERENCES department ("Department_ID")
)
/* Columns in management and all categories for low cardinality columns :
department_ID : 7, 15, 2, 11
head_ID : 5, 4, 6, 3, 10
temporary_acting : Yes, No
*/

CREATE TABLE department (
"Department_ID" INTEGER,
"Name" TEXT,
"Creation" TEXT,
"Ranking" INTEGER,
"Budget_in_Billions" REAL,
"Num_Employees" REAL,
PRIMARY KEY ("Department_ID")
)
/* Columns in department and 3 examples in each column for high cardinality columns :
Department_ID : 1, 13, 11
Name : Energy, Interior, Health and Human Services
Creation : 1913, 1979, 1989
Ranking : 1, 13, 11
Budget_in_Billions : 10.7, 77.6, 59.7
Num_Employees : 112557.0, 3000000.0, 235000.0
*/

...
CREATE TABLE head (
"head_ID" INTEGER,
name TEXT,
born_state TEXT,
age REAL,
PRIMARY KEY ("head_ID")
)
/* Columns in head and all categories for low cardinality columns :
head_ID : 1, 2, 5, 7, 8, 4, 6, 3, 10, 9
name : Jeff Maggert, Pádraig Harrington, Billy Mayfair, K. J. Choi, Dudley Hart, Sergio García,
born_state : Delaware, Connecticut, Alabama, California, Florida
```

```
age : 69.0, 67.0, 68.0, 53.0, 56.0, 52.0, 50.0, 43.0
*/

...
```

The user prompt

The user prompt is simple, the user question for each sample in Spider. For example:

```
How many heads of the departments are older than 56 ?
```

The assistant prompt

The assistant prompt is also simple, containing the associated SQL query from Spider and the reasoning step to find the correct column and correct table for the SQL query. To construct the reasoning step we simply extracted the tables and columns that are used in the SQL query. For example:

```
To construct the query, I'll be working with the following tables: head.
From these tables, I'll be using the following columns: age.
The SQL query I'll be generating is:
SELECT count(*) FROM head WHERE age > 56
```

Submitting the training set for fine-tuning

Once we have created the JSONL file (you can find a small sample here), the next step involves uploading the created file to OpenAI using the following command:

```
openai.api_key = os.getenv("OPENAI_API_KEY")
print(openai.File.create(file=open("spider-finetuning.jsonl", "rb"),purpose='fine-tune'))
```

After uploading the file you can check the status of the upload using the following command:

```
print(openai.File.retrieve(id="file-id"))
#OR
print(openai.File.list())
```

The result should be something like this:

```
{
"object": "file",
"id": "file-id",
"purpose": "fine-tune",
"filename": "file",
"bytes": 71699079,
"created_at": 1693343752,
"status": "uploaded",
"status_details": null
}
```

When the status has changed to processed (similar to below) you can use the file for fine-tuning:

```
{
"object": "file",
"id": "file-id",
"purpose": "fine-tune",
"filename": "file",
"bytes": 71699079,
"created_at": 1693343752,
"status": "processed",
"status_details": null
}
```

Now, we are ready to start the fine-tuning job. To create a fine-tuning job you can use the following python code:

```
print(openai.FineTuningJob.create(
training_file="file-id",
model="gpt-3.5-turbo",
suffix = "spider",
hyperparameters = {
"n_epochs": #number_of_epochs,
})
)
```

The duration of the fine-tuning process will vary depending on the size of the fine-tuning dataset. There is a maximum token limit for fine-tuning, which is set at 50,000,000 tokens. Therefore, when working with the Spider dataset, we reduced the number of samples from 7,000 to 5,750 and conducted fine-tuning for a total of 2 epochs.

You can check the status of the fine-tuning job using the following command:

```
print(openai.FineTuningJob.retrieve(id="ftjob-id"))
```

The result should be something like this:

```
{
"object": "fine_tuning.job",
"id": "ftjob-id",
"model": "gpt-3.5-turbo-0613",
"created_at": 1693346245,
"finished_at": 1693353313,
"fine_tuned_model": "ft:gpt-3.5-turbo-0613:dataherald:spider:id",
"organization_id": "org-id",
"result_files": [
"file-id"
],
"status": "succeeded",
"validation_file": null,
"training_file": "file-id",
"hyperparameters": {
"n_epochs": 2
},
"trained_tokens": 44722020
}
```

Model Performance

We benchmarked the performance of the fine-tuned model against GPT3.5-Turbo without fine-tuning and DIN-SQL + GPT-4 (the current state of the art on Spider) for zero-shot performance.

The results are as follows

Performance of the fine-tuned GPT-3.5-Turbo against previous methods.

Fine-tuning GPT-3.5-Turbo yielded a performance improvement of nearly 11 percent brining its accuracy in line with the DIN-SQL + GPT-4, the current state-of-the-art approach which uses GPT-4 and employs various advanced prompting techniques, including few-shot prompting, chain-of-thought prompting and decomposed prompting.

Critically, the fine-tuned model **significantly reduces both cost and processing time** when compared to the DIN-SQL + GPT-4 approach. The table below provides an approximate cost and speed of difference between the models per single question from Spider.

Cost and speed of different models per question from Spider benchmark

As demonstrated above, the cost of the fine-tuned GPT-3.5-Turbo model is **30 times less** than DIN-SQL with GPT-4 and it is **12 times faster.**

Conclusion and Next Steps

The results from the experiment are clear: with an initial investment of time and money to build a training dataset the state of the art can be matched in accuracy, while being 12 times faster and 30 times cheaper.

Fine-tuning is a powerful tool in the NL-2-SQL arsenal. However it is not a silver bullet as few organizations have NL-to-SQL training datasets readily available. It is our belief that the best architectures will combine fine-tuned models together with RAG agents. With the anticipated launch of GPT-4 fine-tuning, we expect progress in the field to accelerate further and finally unlock question-answering from structured data for all businesses.

In the next post we will show how to plug in the fine-tuned model above into the Dataherald engine and deploy it in a real world scenario.

If you are interested in NL-2-SQL discussions you can join our Discord server. If you want to allow non-technical users to ask questions from your company's data warehouse please join our waitlist.

References

DIN-SQL paper: https://arxiv.org/abs/2304.11015

NL-to-SQL useful papers:

How to Prompt LLMs for Text-to-SQL: https://arxiv.org/abs/2305.11853

Divide and Prompt: https://arxiv.org/abs/2304.11556

Exploring Chain-of-Thought Style Prompting for Text-to-SQL: https://arxiv.org/abs/2305.14215

A comprehensive evaluation of ChatGPT's zero-shot Text-to-SQL capability: https://arxiv.org/abs/2303.13547

Follow

Written by Mo Pourreza

9 Followers · Editor for Dataherald

My email: pourreza@ualberta.ca

More from Mo Pourreza and Dataherald

Mo Pourreza in Dataherald

Evaluating LLM generated SQL

An analysis of current approaches

8 min read · Jul 10

-- 1

Dishen Wang in Dataherald

How to connect LLM to SQL database with LangChain SQLChain

How to Tutorial for using LangChain SQLChain

6 min read · Jun 15

-- 4

Dishen Wang in Dataherald

How to connect LLM to SQL database with LlamaIndex

How to Tutorial for using LlamaIndex for text-to-SQL

5 min read · Jun 30

--

Dishen Wang in Dataherald

How to connect LLM to SQL database with LangChain SQLAgent

How to Tutorial for using LangChain SQLAgent

6 min read · Jun 21

-- 1

See all from Mo Pourreza

See all from Dataherald

Recommended from Medium

The PyCoach in Artificial Corner

Python in Excel Will Reshape How Data Analysts Work

Microsoft just announced Python in Excel. Here’s how it’ll change the way Python and Excel analysts work.

· 5 min read · Aug 24

-- 27

Yash Bhaskar

Introduction to LLMs and the generative AI : Part 1- LLM Architecture, Prompt Engineering and LLM...

Large language models (LLMs) have revolutionized the field of artificial intelligence (AI) development, offering developers unprecedented...

11 min read · Jul 16

--

Lists

Staff Picks
419 stories · 247 saves

Stories to Help You Level-Up at Work
19 stories · 194 saves

Self-Improvement 101
20 stories · 479 saves

Productivity 101
20 stories · 472 saves

Brandon Wohlwend

Mastering SQL Aggregating Functions and Grouping

Welcome back, fellow data explorers! In our previous adventure through the enchanting world of SQL statements, we uncovered the magic of...

16 min read · Aug 2

-- 1

Daniel Liden in The Inner Join

Make ChatGPT Stop Chatting and Start Writing SQL

You are a natural language to SQL code translator. Your role is to translate text to SQL. Return only SQL. Do not include...

8 min read · Mar 8

-- 1

Arwin Lashawn

3 Things I Learned Right After Joining Microsoft as an Azure Support Engineer

I joined Microsoft (NASDAQ: MSFT) back in July 2022 and ever since then I have been consciously keeping track of what I have been learning...

· 5 min read · Apr 9

--

Anwesh Mishra in Dev Genius

Effectively querying any CSV with python, ChatGPT and langchain

Each and every organization who claim to be a “tech” company nowadays need some kind of data analysis to stay ahead in their game. Whether...

5 min read · Jul 5

--

See more recommendations