REPORT

Current Trends in AI (2023-2024)

EL HIHI TAHA
0567414
Taha.El.Hihi@vub.be
Sciences & Bio-Engineering Sciences

Table of contents

1. Introduction

In this report, I describe my experience during my “journey” in the LLM challenge. Many different approaches were tried, from QA databases, rag pipelines to system prompting only.
Instead of the classic “report” format, I tried to present it as a story. A lot of experiments will be presented and explained, and each attempt will be justified before presenting the final version of the solution (section 4.2. and 4.3.). For the final implementation please look in folder “rags”, “Final.ipynb” for the first part and “parser.pl” for the second part.

The final implementation is composed of 2 parts:

- An LLM part to rewrite the query into an easily parsable intermediate query. The idea is to extract all the semantics needed at this stage (e.g., is it a regression or clustering, what is the dependent variables, …).
  This report will cover all the approaches tried to implement this part.

- The second part is a prolog parser that parses the intermediate query to a sequence of AutoML symbols. I wanted to be creative and use declarative programming, using prolog, to implement the parser.

The GitHub repository contains a lot of notebooks, in order to avoid this report being full of code, this report will mention the appropriate files to talk about implementations and their results.
Please also take a look at the demo video, in which some notebooks are shown in detail.

All code is derived from examples provided in the txtai github repository:
https://github.com/neuml/txtai/tree/master/examples

Especially:
https://github.com/neuml/txtai/blob/master/examples/53_Integrate_LLM_Frameworks.ipynb
https://github.com/neuml/txtai/blob/master/examples/52_Build_RAG_pipelines_with_txtai.ipynb
https://github.com/neuml/txtai/blob/master/examples/42_Prompt_driven_search_with_LLMs.ipynb
https://github.com/neuml/txtai/blob/master/examples/34_Build_a_QA_database.ipynb

All approaches use txtai to integrate llama.cpp (the specific model is: https://huggingface.co/TheBloke/Mistral-7B-OpenOrca-GGUF).

2. QA Database

Even if this approach is not the most appropriate for our use case, I wanted to experiment. The idea behind this approach is to find the closest existing question to a user question/ input.

Before starting, we need to set up a QA database based on embeddings. Without going into the theoretical details of embeddings, each question-answer combination is "vectorized" (transformed into numbers :)). Next, the user input will also be vectorized and, using a similarity calculation (e.g., cosine similarity), it will be checked which saved instance is most similar to the user input (and the corresponding answer will be returned).

The txtai documentation has a very interesting section on this.
(https://neuml.github.io/txtai/embeddings/)

Please look for the folder named "QA" for the 2 tested approaches. The second is the same as the first one but in the QA database I specify which element is the example input and which element is the rewritten input. They both gave similar results.

The main disadvantage for this approach, in our context, is that we need a lot of examples in the database for good results. Even if I specified, in the system prompt, that it can slightly adapt the fetched record if needed. It has some problems generalizing for inputs (e.g., see example containing "Train a model....").
Surprisingly, there were some good results, but not enough to consider it a success. Please consult the corresponding notebooks.

3. System/input prompting

The next approach I tried was to optimize the system/input prompt and add a lot of details. All the approaches are present in the "systemPrompting" folder.
I wanted to test whether it was possible to implement the solution for the challenge just by detailing the instructions to the LLM.
A distinction must be made between the system prompt and the input prompt.
During the system prompt, i.e., the moment when you tell the "AI" what its role is and what it has to do and how. The input prompt corresponds to the user's prompt/input, which must correspond to how the LLM was calibrated during the system prompt.
Overall, this approach gave good results but there were 3 major challenges.
The first was to recognize whether the input was about regression or clustering, the second was to ensure that the intermediate query only contained words from the assigned vocabulary, and the third was to ensure that the model recognized the attributes that needed calculation (e.g., add "CalculateVolume" if volume is mentioned).

The general idea of this approach is to explain, in detail, how the model should rewrite the input (which was a lot harder than I first thought).

3.1. First prompt

First, I will discuss the initial approaches, including TXT.ipynb, TXT2.ipynb, TXT3.ipynb, TXT4.ipynb, TXT5.ipynb and TXT7.ipynb.
All these files contain approaches based on the same prompt. The prompt has been continuously adapted to address the challenges.

For example, in TXT3, instead of just asking the model to recognize whether the input refers to a regression or a clustering. I help it and give it possible keywords for the 2 cases (for instance, for a clustering: number of types, variety, identifying types, etc.).
From TXT4, the vocabulary that the model can use when generating responses is explicitly mentioned.
Finally, from TXT7 onwards, I make a distinction between the vocabulary for regression cases and clustering cases to avoid the model using terms specific to regression when it knows that it is a clustering.

Conclusion: the results were quite satisfying, despite containing a few imprecisions (sometimes).

3.2. Second prompt

In this section I will discuss the next approaches, including TXT8.ipynb and all notebooks starting with “TXT9” (hence, also notebooks such as TXT93 and TXT991). A lot of notebooks are very similar, so maybe I could have removed a few, but I wanted to keep every new detail added or modified to the initial prompt.

In this second way of prompting, I tried a new approach. In the first approach, the prompt contains a text in "tutorial" format. In this second approach, text in "instructions" and "sections" format have been used.
For transparency's sake, all prompts after TXT8 are modifications of TXT 8.
And the TXT8 prompt was generated by prompting GPT-4.
I wanted to experiment generating a prompt for an open-source model by prompting another model. And the idea of structuring the prompt impressed me and gave better results.

When I mention that this method generates better results, it means that it is able to distinguish perfectly between regression and clustering. Moreover, with a few rare exceptions, the model uses only the specified vocabulary.
The only problem that remains, and that has required changing the prompt a dozen times without success, is recognizing when to add the calculation of an attribute.

Some examples:

In TXT8, the following is present: Include "calculate volume," or "calculate density," if the input specifies volume or density.
An instruction that is easy for humans to understand, but which the model sometimes ignores.

In TXT9, I experimented with reversing the instruction and starting with the condition first: Only if the input specifies volume or density, include "calculate volume," or "calculate density,".
Sometimes, a small change in the prompt can have a very positive effect.

In TXT95 I attempted to make things easier for the model and to make a distinction between complex attributes and simple attributes (and at the same time take attribute names out of the vocabulary and put them in a separate section):

```
# Attributes:

- **simple-attributes:** id, surface, height, mass.

- **complex-attributes:** volume, density.

2. If the input mentions a complex-attribute, include "calculate [attribute],". #
```

In TXT97, I removed volume and density of vocabulary and try to instruct the model on unknown attributes:

```
Attributes: id, surface, height, mass.

2. If the input mentions an unknown attribute, include "calculate [attribute],".
```

4. Rag (Retrieval-Augmented Generation)

4.1. Original idea, generating intermediate query

The idea behind RAG is to feed the prompt only with the context that is relevant to our input. To explain with our use case, whether the input is a clustering or a regression, in the system prompting section I feed as context the 2 whole lists of instructions. This can lead to imprecision and errors when generating the response. Another advantage is that less computation power is needed, since the context has become smaller.
One possible approach is to "split" the context by document or even by paragraph. It is therefore possible to use information from several documents at the same time, without having to load the entire 2 documents.

This approach is also based on generating embeddings from the whole context and calculating similarity with the input to see whether a part of the context is relevant to our input.
For this section, please refer to the "rags" folder.

The first test is present in TXT6 (see context6.txt for corresponding context). The idea was to have common instructions for clustering inputs and regression inputs, and to leave 1 paragraph explaining the entire procedure for regression and 1 paragraph for the entire procedure for clustering. Normally, during the similarity check, the appropriate part should be added to the context.

The results were sometimes good, but some responses contained a lot of useless words not present in the vocabulary and were sometimes incomplete.

The next try is in TXT993, this time attempting to build and use 2 different context files. The first containing the instructions (in a structured way) for inputs indicating clustering and the second for inputs indicating regression.
The idea is that now that contexts have become smaller and more concise, without diminishing the information needed. I was able to make some space and add examples of inputs and expected responses with the hope of improving the quality of responses (see clustcontext993.txt and regcontext993.txt in the rags folder).
Even if the model respects the vocabulary perfectly, and recognizes perfectly what kind of instructions it should follow (clustering or regression).
Unfortunately, it still has a bit of trouble, sometimes, with recognizing when it should add an attribute calculation command or not.
As can be seen in the notebook, there is one case where it adds "Calculate mass", the mass attribute does not need any calculation.

4.2. New and final idea: semantic extraction without explicitly generating intermediate queries as before.

Arriving at this point, I started brainstorming. Maybe I focused too much on the intermediate query format and not enough on other options.
Now, the question is, what is the purpose of intermediate query? The point is to extract the information we need in a more parsable format than the initial input.
I was thinking about whether I really need to add information such as "calculate volume" at this stage. For instance, assume the following input: "Predict height based on volume through regression". In our context, we can simply extract the attribute names and the type of input (i.e., volume and mass, and check if it is a regression or not).
If I have this information, I can easily generate the corresponding AutoML sequence using a parser. And without having to go through an intermediate query that follows a very (too?) precise format.

The idea was therefore to use RAG to retrieve the appropriate context / instructions to add to the prompt (regression or clustering). In addition, the instructions will focus more on extracting the necessary semantics than on generating the intermediate query (as before).
The latter has been implemented in TXT994 (possible contexts are regcontext994.txt and culstercontext994.txt). A comparison of the 2 prompt styles in the case of regression:

Previous style, focused on generating the intermediate query (regcontext993.txt):

```
# Context:

- Use regression-vocabulary for inputs involving model training, regression, prediction or estimation.

- Simple attributes: id, mass, height, surface.

- Complex attributes: volume, density.
```

# Vocabulary:

- Regression-vocabulary: load, dataset, calculate, regression, data, perform, on, calculate, using, split, MSE.

# Instructions:

1. Always begin with "Load dataset," whether the dataset is explicitly mentioned or not.
2. If a complex attribute is mentioned in the input, follow with "calculate [complex-attribute(s)],".
3. Follow with "split dataset,"
4. Follow with "perform regression on [dependent attribute] using [independent attribute(s)],"
5. Conclude with "calculate MSE."

# Example input and expected response:

- Example with 1 complex attribute:

Input: "Predict volume based on height and mass through regression analysis."

Expected response: "Load dataset, calculate volume, split dataset, perform regression on volume using height and mass, calculate MSE."

- Example with multiple complex attributes:

Input: "Predict volume based on density and surface through regression analysis."

Expected response: "Load dataset, calculate volume and density, split dataset, perform regression on volume using density and surface, calculate MSE."

- Example without complex attribute:

Input: "Predict surface based on mass through regression analysis."

Expected response: "Load dataset, split dataset, perform regression on surface using mass, calculate MSE."

New style, focused on extracting the needed semantics (regcontext994.txt):

You are an AI tasked with identifying attributes from input sentences that describe regression modeling tasks, predictions, estimations, determinations using attributes, or model training.

Follow the instructions to generate the response.

# Context:

- Possible attributes: id, mass, height, surface, volume, density
- You will determine which attribute is the target of prediction (dependent), and which attributes are used for making predictions (independent).

# Instructions:

1. Always begin with "Regression,".

2. Follow with "dependent attribute: [attribute],", which is the attribute that is being predicted.

3. Follow with "independent attributes: [attributes].", which are the attributes used to predict the dependent attribute.

As explained above, the idea is to delegate everything possible to the parser. In our use case, the only information we need to obtain at the LLM stage is:
1) what type of task it is (regression or clustering) (solved by RAG)
2) Which attributes are used and what are their roles (dependent or independent)?
These 2 tasks are suited to LLMs!
We do not need to prepend "Load Dataset" or add "calculate Volume". Since we are supposed to know our datasets, we can handle attribute manipulation at the next stage (at parser level) and focus on what we need.
Moreover, this approach is scalable. Even if, for the moment, we only test regression models using MSE. The prompt can easily be changed to include the extraction of the way the model is tested, or a new type of task can be added by adding a document.
As can be seen in the TXT994 notebook, the results are perfect (for the chosen examples).

4.3. Final pipeline

The final pipeline can be found in Final.ipynb (folder 'rags'). It consists of the LLM part of TXT994, a small cleaning phase (to remove \n's for instance) and the parser part. The parser is written in prolog and can be found in parser.pl in the 'rags' folder. Prolog is a logic programming language. To keep it short, a Prolog program is a database of facts (knowledge base) and rules (predicates) that describe relationships between variables. It solves problems by querying this database and deriving conclusions logically (unifying variables) (reference: Declarative programming course at VUB).

The code is well documented, but the general idea is that this parser, written in Prolog, analyzes the output of our LLM and generates the appropriate sequence of AutoML symbols.
There are two outputs, the sequence of AutoML symbols and a list corresponding to the elements to be fed to an operation.
Both lists are of equal size, so we need to look at the corresponding position to find the input for each "command". For instance, calculateVolume will always have [surfaceInSquareMeters,heightInMeters] as input.

To configure the parser, I assume that the user knows the datasets he is working with.
For example, the task_type predicate, currently has two possible values.
If another task type is given as input, it will not be able to parse it.
task_type(regression) --> [regression].
task_type(clustering) --> [clustering].

As another example, the is_attribute predicate has a known list of attributes, and there is another predicate that knows which attributes are complex (i.e. require calculations).

```
is_attribute(Attr) :- memberchk(Attr, [id, surface, mass, height, volume,
density]).
is_complex(Attr) :- memberchk(Attr, [volume, density]).
```

And of course, the parser is scalable, and we can add new datasets and corresponding attributes if required, but for now we are working with only one dataset.

Last example present in Final.ipynb:

```
llm_to_AutoML_translator("Train a model that can estimate density based on volume and mass given the objects dataset")
```

```
intermediate:
Regression, dependent attribute: density, independent attributes: volume, mass.



AutoML Symbols:
[loadData,calculateDensity,calculateVolume,splitTrainTestData,knnRegression,regressionMSEError]
Data columns:
[[],[massInKilograms,surfaceInSquareMeters],[surfaceInSquareMeters,heightInMeters],[],[[volume,mass],density],
knnRegression_OUTPUT]
```

As shown in the example above, after "intermediate" we can see the output of the LLM part: the semantics we need have been extracted.
It is known that this is a regression and the attributes used and their roles (dependent or independent) are known.
The parser parses correctly and generates the appropriate AutoML sequence. It will also generate a list containing the element(s) to be fed to each operation.
For instance, "loadDataset" needs no input. Since we only work with one dataset, a possible input could be the name of the dataset.
The parser knows, thanks to its knowledge base, that we need to calculate density and volume, since these are attributes marked as complex. The elements to be fed to calculateVolume are, [surfaceInSquareMeters,heightInMeters] (again, thanks to our knowledge base).
The input to knnRegression is a list of 2 elements, the first element is a list of independent attributes and the 2nd element is the independent attribute (in our case, [volume,mass],density],).
The input of regressionMSEError will simply be the output of the previous phase (knnRegression_OUTPUT).

5. Conclusion

I am satisfied with the results of this challenge and how I worked on it. I learned a lot during this challenge, and I will not stop learning about LLMs. I am very happy to have chosen this challenge!