Natural Language Processing Assistant - Working Notes

Mikoláš Fromm

29.7.2023

1 Introduction

The goal of this project is to create a program that will be able to process natural language queries and translate them into SQL-like language, which will then be executed on a database. The main goal of this project is to work with a natural language and find the best way to parse it with today's favorite language models into custom-made query language. In order to achieve it, an OpenAI ChatBot API will be used to transform the language into a well-defined syntax which is necessary to execute the query.

2 Basic approaches

At the beginning before executing any transformation, a user is prompted to enter a whole command describing what to achieve as a result. This command is entirely in a natural language form, for example:

> *Sort it by the name, drop those with age under 18 and group them by their gender.*

which should be translated into something similar to:

> SortBy."Name".FilterBy."Age".>=."18".GroupBy."Gender".

As you can see, the form of the *executable* command is still very well readable, allowing the user to step into forming such a command in the middle of the process and continue by its own.

Every transformation has a defined output type, which determines what following operations are executable. Therefore, when creating such a command, the application suggests a list of all available transformations and its optional arguments after each step. For example after submitting *SortBy* it will suggest *Name* and *Age* as possible arguments. The language processing assistant should be therefore capable of forming the final command entirely by itself, but also allow the user to step in and just whisper the missing (and best fitting) parts of the command.

2.1 Matching words

Since every transformation has its unique name, we might try to ask the LM to return exactly one name of a transformation from a given collection which is the closest to the next step to form the final result. This is the simplest approach, but it is also the most error-prone. The LM should be able to return the correct transformation name in most cases. However, it is difficult sometimes to convince the LM to answer only a single word from the given collection, especially when the collection does not contain perfectly fitting option. Let me illustrate it on an example, where the user input was as follows: *Sort by name and group by hometown.*

First collection: *SortBy, FilterBy, GroupBy, DropColumn*

LM choice: *SortBy*

Second collection: *FirstName, LastName, Age, Hometown (content dependant)*

LM choice: *name*

ERROR: *Invalid input, you must choose from the options given above. Please try again.*

LM choice: *Sorry about that. Here are the options again:*
$> FirstName$
$> LastName$
$> Age$
$> Department$
$> Gender$
$> HomeTown$

As shown in the example, when the LM mismatches the answer, it starts generating longer answers which are intended to help the user to form the correct answer. However, this is not the desired behavior, since the user is not supposed to be asked twice for the same thing, nor does the user need to get his options repeated. Most importantly, in the fully autonomous mode, where only the LM forms the final query, this is the dead end. The LM should be able to return the correct answer in the first place.

Sometimes it also happens that the LM returns a word semantically similar to what is contained in the collection, but which differs syntactically. For example, when the user input is *Sort by name and group by hometown.* and the collection is *FirstName, LastName, Age, Hometown*, the LM might return *First Name* instead of *FirstName*. This also is an error behavior, since the *word-matching* approach requires the LM to return exactly the same word as it is in the collection.

2.2 Indexing

Therefore, it seems promising to index each option in the collection of next moves and ask the LM to only pick the best fitting option and return its index. This way, the LM is not forced to return the exact word, which might eliminate many potential errors.

Sadly, this approach does not help when the LM is asked to return a real word which is not straightly related to the database. For example the usage of *FilterBy* requires two arguments: *relation* and *right side of the relation*, where the *right side* might be any meaningful word, for example:

FilterBy.Age.>=.18.

Therefore, the LM is not able to return the index of the *right side* argument, since it is not contained in the collection of possible options.

Unfortunately, even this approach needs tweaking of the permanent system messages passed to the LM at the beginning of the conversation. When the LM is introduced to its role by the following messages:

1. *You are an assistant who should translate given user input into a query request.*
2. *You always get instructions and options from which you can choose.*
3. *You write your choice as a number from the [] brackets. Dont write anything else!*
4. *Only when you are asked to write a word, you can write any word you want.*
5. *Dont ask any questions or dont give any following options. Just answer.*

the LM responses to the previous request looks like:

First collection: *[0] SortBy, [1] FilterBy, [2] GroupBy, [3] DropColumn*

LM choice: *Please choose [0] SortBy*

ERROR: *Invalid input, you must enter only an integer. Please try again.*

LM choice: *Please choose [0] SortBy*

And so on...

Not only the LM does not answer in the introduced format, but also is unable to recreate the answer to match the right format, even though the error message clearly states what is the expected format. It gets even more exciting (and useless) when we reduce the number of the introducing system messages at the beginning of a conversation. If we only specify how the LM answer should look like, leaving only the messages 3. and 4. from the previous list, the LM responses to the previous request looks like:

First collection: *[0] SortBy, [1] FilterBy, [2] GroupBy, [3] DropColumn*

LM choice: *Let's start with SortBy. Please choose from the following options:*
[0] Ascending,
[1] Descending

which is absolutely out of the format, even though part of the response contains the important keyword.

The next example shows that the initial introducing messages matter a lot. When we give the LM the following instructions:

1. *You are an external API which translates user input into a query request.*
2. *You sequentially build the query from left to right. You always get all next possible actions and you must always choose one.*
3. *You always get a list of all available transformations or its arguments. When one transformation is finished, you get all possible next transformations.*
4. *If you feel you have finished the query, choose the [0] Empty transformation.*
5. *You must write only the numbers from the brackets. Dont write anything else!*
6. *If the answer should not be a number, write the whole appropriate word.*
7. *Dont ask any questions or dont give any following options. Just answer.*

the LM responses started to match the required format. However, the LM was able to match only the last requested transformation from the user input and forgot to include the previous ones.

In more tests with different user requests, the LM performed very well when the user input was only one transformation request, for example: *Sort the people by their age* or *Filter people with hometown Praha*, but even a combination of two successful requests resulted in a failure, for example: *Sort the people by their age and filter those with hometown Praha.* LM was able to match only the last transformation, or failed to create any of them.