An ad-hoc Learned Natural Language Interface for Databases

Vasileios Liakopoulos
vliakop@outlook.com
National and Kapodistrian University of Athens

Kapetanas Nikolaos
nkapetanas@outlook.com
National and Kapodistrian University of Athens

Abstract

This paper demonstrates the approach and implementation decisions followed for a natural language interface for a specific database schema. Initially presented in DBPal: A Learned Natural Language Prasetya Utama et al[3], we follow the pipeline proposed, slightly differentiated in terms of the range of SQL queries supported.

Keywords: natural language, neural networks, databases, interface

ACM Reference Format:
Vasileios Liakopoulos and Kapetanas Nikolaos. 2020. An ad-hoc Learned Natural Language Interface for Databases. In *Proceedings of Database Systems (DSIT).* Athens, Greece, 5 pages.

1 Introduction

The ubiquity of databases in our lives has led to an increased need of professionals who can process data, manage and query database systems. Most tasks are supported by the domain specific language Structured Query Language, widely known as SQL. However robust and expressive SQL is, its use is mostly limited to those with a computer science background. The steep learning curve of SQL does not lend itself to scientists or everyday users, e.g. professionals in healthcare or retail, who need to access data to easily learn and use the language.

For that reason and thanks to the advancements in Artificial Intelligence and Natural Language Processing the recent years, Prasetya Utama et al[3] propose a Natural Language Interface for Databases, enabling simple users to extract data from databases through natural language queries, requiring no prior knowledge of SQL.

The paper is structured as follows: In (2) we present an overview of the architecture of Prasetya Utama et al[3] . In (3), we present our approach and decisions in our attempt to develop a similar Natural Language Interface for a specific Database. In section (4), we present an evaluation of our system through the experiments and a small-scale user study conducted. Finally, in (5) we discuss our overall thoughts and discuss future improvements.

2 DBPal: A Learned Natural Language Interface for Databases using Distant Supervision - Overview

As noted in the introduction, the motivation of Prasetya Utama et al[3] was the easy access of users to retrieve data stored in databases. The Natural Language Interface for Databases they developed consists of two parts: the user side and the server side.

2.1 User Side

The user side of the system consists of a simple form in which the user types their query in natural language, e.g. "Show me the population of Athens". The question is then submitted to the server side, where it is processed and a response is returned. The response is a tabular representation of the records matching the criteria specified by the user.

2.2 Server Side

The server side, also mentioned as backend hereafter, involes four parts: The Query Preprocessor, the Neural Translator, the Query Postprocessor and the database where the data are stored.

Query Preprocessor: The Query Preprocessor is responsible for handling constant in the input queries - the queries submited by the users. After the constants have been detected, they are replaced with placeholder in an attempt to reach a generalised form and improve the accuracy of the Neural Translator. For example, given as input query the sentence "Show me the patients diagnosed with covid-19", the term 'covid-19' is replaced with '@DISEASE' leading to the sentence "Show me the patients diagnosed with @DISEASE".

Query Postprocessor: The Query Postprocessor is responsible for the reverse functionality of the Query Preprocessor. After the Neural Translator has matched the input query to an SQL query, every placeholder is replaced by the initial value submitted by the user. Following the example above, "Show me all patients diagnosed with @DISEASE" would match to "SELECT * FROM patients WHERE diagnosis = @DISEASE" which in turn would lead to "SELECT * FROM patients WHERE diagnosis = 'covid-19'".

Neural Translator: The Neural Translator is a Sequence-to-sequence neural network whose functionality is to translate the natural language statements to its SQL counterparts.

The interest about the Neural Translator focuses on the training set used to train the model. The complexity of databases - different schemas lead to different SQL queries - account for the lack of datasets in which SQL queries are mapped to natural language statements. In order to tackle this issue, a training dataset was build from scratch, based on the following steps.

2.2.1 SQL-NL pairs template: The starting point is the notion that every SQL query for a given SQL schema, can be mapped to a natural language sentence and therefore a set consisting of such mappings can be created. For example, "SELECT names FROM patients WHERE diagnosis="covid-19" can be mapped to "Show me all patients with covid-19".

2.2.2 Data Augmentation: Because a dataset comprised of each SQL query mapped to a sole natural language statement would not suffice for a training dataset, the Data Augmentation step allows for, what the name suggests, the increase of the training dataset volume. In order to achieve such increase, each word in the initial natural language statement is cloned and then paraphrased, using the Paraphrase Database Ellie Pavlik et al [2]. For instance the sentence "Show me all patients with covid-19" can be paraphrased to "List all patients with covid-19" and "Display all patients with covid-19".

2.2.3 Lemmatizer: In order to improve accuracy of the Neural Translator, each natural language statement is reduced to a more crude form by applying lemmatization to each word of the sentence. For instance, "Show me all patients with covid-19" turns into "Show me all patient with covid-19".

3 A similar approach to DBPal

Our approach of building a Natural Language Interface for Databases is heavily inspired by Prasetya Utama et al[3] .The following section reiterates the architecture and workflow of Prasetya Utama et al[3] with the differences and variations introduced by our implementation. Our system focuses on a single-table database with information about patients. We consider that professionals in healthcare would be the target group of the application. Therefore, many ad-hoc solutions are introduced in this approach.

3.1 User Side

As described in section 2.1, we develop a similar, simple user interface (front end) where the users can be submit their queries. The functionality of the user side remains the same.

For the front end, Angular 8 with NodeJS was used. NodeJS is a JavaScript runtime built on Chrome's V8 JavaScript engine for building and serving the User Interface(UI) part locally or in the production environment. The main component that was created is the home component which has a search input for our query, an empty table to be filled with result data, a label for the SQL response from the Preprocessor, one from the Sequence-to-sequence translator and one as the final query that was generated. When the user clicks the submit button, a GET request is produced for the server side which is implemented in Django Rest Framework.

3.2 Database

In order for the database to be available for the back end, a docker container was used.

Docker is an open source platform as a service (PaaS) product that uses Operating System(OS)-level virtualization to deliver software in packages, called containers.

The containers are isolated from one another and bundle their own software, libraries, and configuration files. All containers are run by a single operating system kernel and therefore use fewer resources than virtual machines.

Our container runs the Linux alpine version, in which a MySQL 5.7.30 version is installed. To configure the docker container, a docker-compose file was used, which downloads the MySQL version, configures it, and automatically initializes it with the DB schema and the data from a SQL file that is provided. The data are locally saved to a volume that is named mysql-data. If any problem is encountered during this process, the docker container automatically restarts.

3.3 Server Side

For the back end, Django Rest framework was used as an API. The architecture of the API is the basic rest layer (views) where the model of the database is use paired with the serializer of the model and the process query service. The process query service is responsible for the preprocessing, translation, postprocessing of the inpurt sentence and finally for the submission of the custom raw SQL query to the database.

When the API receives a GET request from the front end, it forwards the input sentence to the preprocess query method, where first is cleaned from unnecessary things like HTML5 tags and also is lemmatized by the NLTK porter stemmer as we want everything to be in the root of the word.

Then, the cleaned lemmatized sentence is given to the preprocessor, to replace the constants. For replacing the constant values with placeholders like @NAME, @AGE etc., the input sentence is split it into words and for each word we do a lookup to find if it is included as a value in any of the constant lists.

The constant lists are lists that have lemmatized unique words taken from the database records. For example, if a word exists in the field of the first names, that word is replaced by the placeholder of the column, @NAME. Also, the replacement is added to a dictionary that will be used from the postprocessor to replace back the words that were replaced.

Next, the user's input is given to a method that replaces any numeric values with either the placeholder @AGE or

@LENGTH_OF_STAY. Then, the result is passed to the Sequence-to-sequence model which will translate it to SQL with constants (where applicable). That input will be given to the Postprocessor, which will replace the placeholders with the key value pair of the dictionary of the replacements which was created in the preprocessing phase.

Finally, a custom raw query will be executed in the database. The raw query must have the primary key of the table (id) because we are using the mechanism given by Django that executes the raw query and maps the results to patient objects. Those patient objects are serialized and sent to the UI as JSON objects along with the results of the preprocessor layer, the translated query and the postprocessor result (final query form).

Query Preprocessor: The functionality of the Query Preprocessor also remains the same. Every constant in an input query is replaced by a placeholder value.

First, a file is loaded with the data that the generator produced to preprocess them. The data are supposed to be already lemmatized. Every line of the file is English sentence together with a SQL sentence.

For example:

```
show all patient?        SELECT id FROM patients
```

For every line, we split the lines into pairs and for each pair, the words are normalized, by converting the letters to lowercase, removing punctuation and keeping letters, numbers and symbols like "!, _, @, ?". Then, two language objects are created, one is the English language statement and the other is the SQL language query. For each of them, three dictionaries are created since we want to represent each word as a one hot encoding. Therefore, there is a mapping between a unique index → word, word → index, as well as a word → count. The dictionaries are initialized with the word "unknown", which is used later to replace input words of the user that are not existing in the dictionaries created in the training process. Moreover, "EOS", "SOS" words are added in each dictionary that mean end of sentence and start of sentence, respectively. The maximum length of words in each sentence that is allowed in the training dataset is 20.

Query Postprosessor: Once again, no changes have been made in the task that the Query Postprocessor is responsible for. After the Neural Translator has mapped an input query to a SQL query, every placeholder value is replaced by the corresponsing constant.

Neural Translator: In our case, the Sequence-to-sequence neural network is used. However, although heavily inspired by Prasetya Utama et al[3] in the matter of generating the dataset, some slight variations appear in the process.

The translation from English to SQL, was done with a sequence to sequence model.

For example:

```
[KEY: > input, = target, < output]

> = SELECT id, age FROM patients WHERE name = ‘Baker’
< SELECT id, age FROM patients WHERE name = ‘Baker’
```

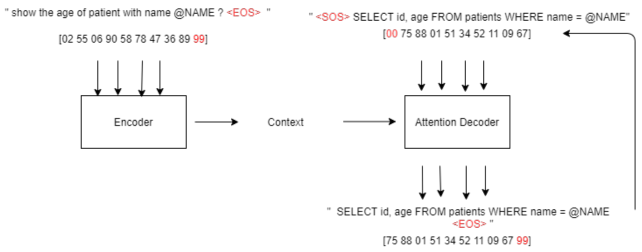

" show the age of patient with name @NAME ? <EOS> "
[02 55 06 90 58 78 47 36 89 99]
Encoder
Context
Attention Decoder
" <SOS> SELECT id, age FROM patients WHERE name = @NAME"
[00 75 88 01 51 34 52 11 09 67]
" SELECT id, age FROM patients WHERE name = @NAME <EOS> "
[75 88 01 51 34 52 11 09 67 99]

Figure 1. Neural Translator Architecture

Seq2Seq model: In the Sequence-to-sequence model, two recurrent neural networks are used to transform one sequence to another.

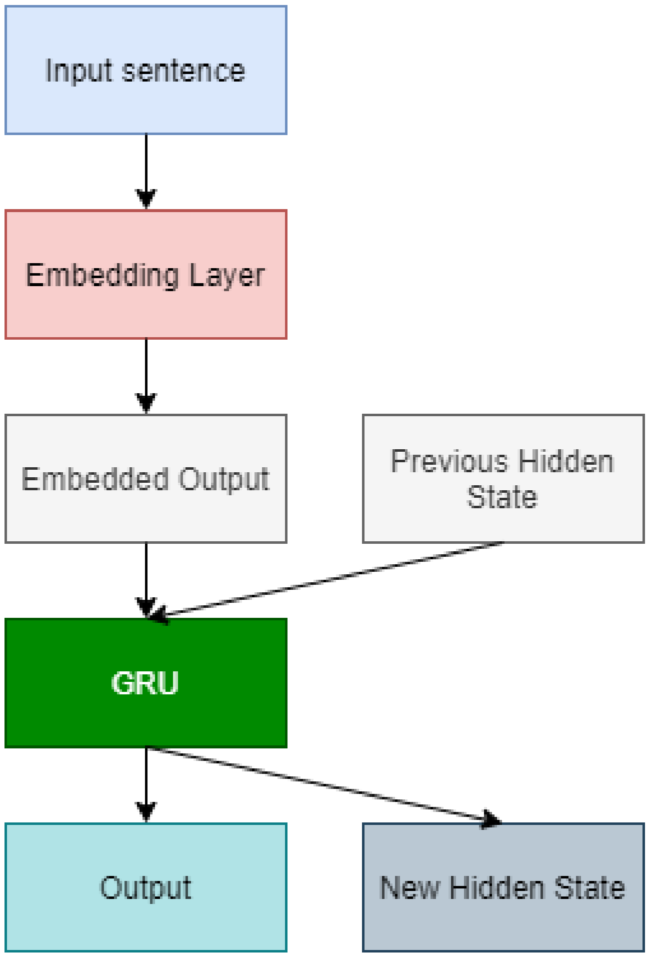
There are two main parts: the encoder and the attention decoder. The encoder transforms the sequence to a more compressed one, in a vector, and the attention decoder unfolds that vector into a new sequence. The decoder uses an attention mechanism, which lets the decoder focus to a specific range of input sequences.

In other words, the decoder decides which parts of the input sentence to pay attention to. By letting the decoder to focus on specific parts of the sentence, the encoder does not need to encode all information in the input sentence into a fixed length vector. The free us from sequence length and order, which makes it ideal for translation between two languages.

For example, consider the two sentences: "give me the age of all patients in the hospital" → "SELECT age FROM patients". Most of the words in the input sentence have not any direct translation, so there are more words than the translated one. Also, the order can be different like "from all the patients of the hospital, give me the age" → "SELECT age FROM patients".

Encoder: In the encoder of Figure1, for each input word, an output vector and a hidden state is produced. The hidden state is used for the next input word.

Attention Decoder: As previously mentioned, the attention allows the decoder (Figure1) to focus on different parts of the encoders output for every step of the decoder's own outputs. First, the attention weights are calculated. These weights will be multiplied by the encoder output vectors to create a weighted combination. The result will contain all information about that specific part of the input sequence and therefore help the decoder choose the right output words. The attention weights are calculated by a feed-forward layer, using the decoder's input and hidden state as inputs. Because there are sentences of all sizes in the training data, to create and train this layer we must choose a maximum sentence



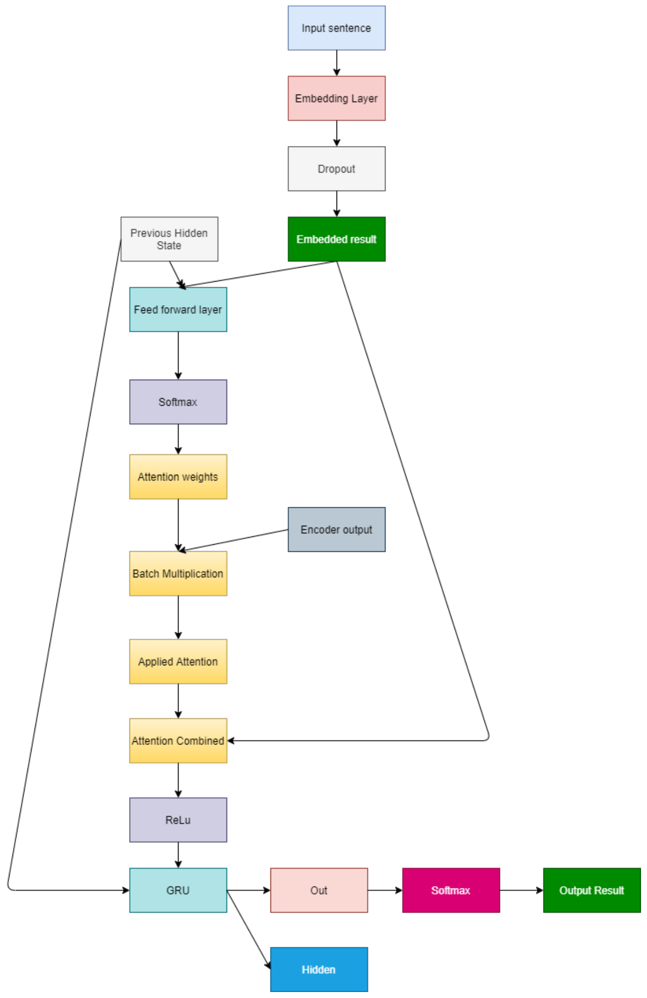
length (input length, for encoder outputs) that it can apply to. The maximum length of the sentences will use all the attention weights, while shorter sentences will only use the first few.

Training Phase: For the training phase, each pair of sentences (one in English and one in SQL), will need an input tensor that has the indexes of the words of the input sentence. While we create those vectors, we append the EOS token to both sentences.

For every input sentence that is feed it in the encoder, the output and the hidden state are saved as there are needed. Then, the decoder appends the SOS token as its first input in the sentence, and the last hidden state of the encoder is used as the first hidden state.

Furthermore, the method of “teacher forcing” is used, where the real target of output is used as the next input, instead of using the decoder’s guess as the next input. This causes the model to converge faster.

Evaluation: The evaluation phase is almost the same as the training but without any targets, and so we feed the decoder predictions back to itself for each step. When it predicts a word, we add it to the output string, and if it predicts the EOS token, we stop and return the values concatenated.



In case where the user uses unknown words in the search input, those words are replaced by the word “unknown”, otherwise we will get an exception.

3.3.1 SQL-NL pairs:

Our approach supports the following 3 categories of SQL queries:

- FROM-queries: All SQL queries with no WHERE-clause in the form of "SELECT <fields> FROM patients". The <fields> can be a single field or any combination (the powerset) of the table columns.
- WHERE-queries: SQL queries with up to 2 equality conditions in the WHERE clause. They come in the form of "SELECT <fields> FROM condition1 (AND condition2)". The <fields> can once again be a single field or any combination (the powerset) of the table columns.
- Aggegators-queries: A list of SQL queries using aggregators such as MIN, MAX, COUNT, AVG that would make sense in a real-life situation, avoiding redundant SQL queries that would generate noise during the training phase of the Neural Translator.

The first difference lies in the fact that for each SQL statement is mapped to 2 natural language sentences. For the first two groups (FROM-queries and WHERE-queries) the generation of both the SQL queries set and its corresponding natural language statements set is generated automatically. For the last group, the Aggregators-queries, both the SQL queries and its natural language mapping is a manual endeavour.

In order to generate the dataset, the following steps were followed:

- A list of the table columns was created
- Every column of the table was mapped to a placeholder value in a python dictionary
- The powerset of the first list was calculated. The powerset is the set of all possible combinations of a set.
- An SQL template (string) and 2 natural language templates (strings) were used
- For the FROM-queries, for every set in the powerset, the from-clause of the SQL template was replaced by a comma seperated list of the elements in the set and accordingly in the two natural language templates
- For the WHERE-queries, the same process as above was followed for the FROM clause. For the WHERE clause, only the sets in the powerset with 2 elements were kept. For every set of size 2, the where-clause was replaced by a list of equality conditions of the elements of the set. The natural language statements were handled accordingly.

3.3.2 Data Augmentation: In this step, each natural language statement is duplicated as many times as the synonyms[1] for every non-stop word in it. Every sentence was broken into words using the NLTK library. The Paraphrase Database Ellie Pavlik et al [2] is used for the paraphrasing process. The stop word list consists of words with high use frequency such as "and", "or", "w-" words which usually do not need paraphrasing and thus avoiding the generation of noise in our training dataset.

3.3.3 Lemmatizer: In the final step, the NLTK porter stemmer is used to bring the natural language sentences to a raw form.

4 Evaluation

For the evaluation of the Natural Language Interface for a database containing patients' information, a user study was conducted.

The schema of the database was explained to the participants, i.e. the kind of data contained in the database, as well as the limitations. Then, they were asked to come up with questions in the English language and submit them.

Totally, 29 questions were submitted, out of which 10 were classified as FROM-queries, 14 as WHERE-queries and the rest as AGGRREGATOR-queries.

The overall success rate was 34%, meaning that 10 out of the 29 input queries in English were 'translated' to the correct SQL query.

The FROM-queries category success rate was 60% of the questions submitted. For the WHERE-queries category the success rate was 29% of the questions asked. In the case of the AGGREGATOR-queries posed, none was translated correctly.

5 Future Work

The main future improvements revolve around the support of more complex SQL queries, as well as more expressive natural language sentences.

The introduction of more operators and conditions in the WHERE clause of the SQL statements lead to a massive training set in terms of volume. More specifically, by incorporating equality conditions for all field combinations, a paraphrased training dataset of 15GB size is produced, posing a major bottleneck to the training of our machine learning model.

To tackle the problem of paraphrasing, we used a single algorithm: Replace every word in the user input sentence with a synonym. Repeat this process for every synonym of that word. This is called the lexical paraphrasing. However, by using more advanced and state of the art natural language techniques for syntactic and phrasal paraphrasing, allowing for a more robust and expressive dataset from the English language perspective.

References

[1] Erick Rocha Fonseca. 2018. *PPDB*. https://github.com/erickrf/ppdb

[2] Ellie Pavlick and Chris Callison-Burch. 2016. Simple PPDB: A Paraphrase Database for Simplification. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Association for Computational Linguistics, Berlin, Germany, 143–148. https://doi.org/10.18653/v1/P16-2024

[3] Prasetya Utama, Nathaniel Weir, Fuat Basik, Carsten Binnig, Ugur Çetintemel, Benjamin Hättasch, Amir Ilkhechi, Shekar Ramaswamy, and Arif Usta. 2018. An End-to-end Neural Natural Language Interface for Databases. *CoRR* abs/1804.00401 (2018).