Translating Natural Language Queries into SQL

Zayd Patel

2020-05-13

Contents

Preface

Summary

This paper takes a look at translating natural language queries into SQL queries in a domain specific environment; Retail. We look at an analytical and deep learning approach to this problem on a custom dataset with question deliberately written in such a way to test the approaches with a wide variety of SQL queries that use many different SQL operators. This paper found that for this specific domain, an analytical approach is not only more accurate, but less resources intensive and is more customisable making it easier to embed into other application

0.1 Acknowledgements

- This document is written using bookdown (rstudio, 2020) as well as various R packages that Bookdown depends on.
- This document use a CSL style called `Cite Them Right 10th edition - Harvard` which is saved as the file `harvard.csl` (Zotero, no date).
- Project used Python 3.7 (Python Software Foundation, no dateb) and various other python packages which will be cited throughout the paper.
- Diagrams drawn with draw.io ('Jgraph/drawio', 2020) and/or plantuml ('Plantuml/plantuml', 2020).
- Program of Study: **BSc (Hons) Computer Science**.
- Project Supervisor: **Dr Juilan Hough**.

0.2 Abbreviations

- **SQL**: Structured Query Language
- **NLU**: Natural Query Understanding
- **CLI**: Command Line Interface
- **HTML**: HyperText Markup Language
- **ORM**: Object Relational Mapper
- **ERD**: Entity Relationship Diagram
- **LSTM**: Long Short Term Memory
- **API**: Application Programming Interface
- **REST**: REpresentational State Transfer
- **JSON**: JavaScript Object Notation

0.3 Definitions

- **Database**: A structured set of data held in a computer that is accessible in various ways.
- **ORM**: Object/Relational Mapping (ORM) provides a *methodology and mechanism for object-oriented systems to hold their long-term data safely in a database, with transactional control over it, yet have it expressed when needed in program objects* (O'Neil, 2008).
- **ERD**: An ERD is a *data model used to describe the relationship between different entities including the different attributes of each entity* (Chen, 1976).
- **Reinforcement Learning**: A type of Artificial Intelligence where by artificial agents learn for themselves through trial and error by being rewarded for successful strategies and punished by unsuccessful strategies (Silver, 2016).
- **Neural Network**: Neural Networks are a way of doing machine learning by analysing hand labeled training examples. Neural Networks are loosely modelled on the human brain as they consist of nodes, usually organised into layers, that feed into other nodes. Each individual node is adjusted until the final layer of nodes, also known as the output layer, correctly outputs data as expected in accordance to the labeled training data. (Hardesty, 2017).
- **Relational Database**: A relational Database is type of database where the data is organised into tables which can then be linked or *related* to each other. For example, you can have a table with customer details and a table with order details and you can implement a relationship between those two tables and extract new pieces of information - for example, you can run a query which shows all the customers that have ordered more than 5 products.(IBM Cloud Education, 2019)
- **Stop Words**: Stop words are words that appear to be of little value and are excluded from the vocabulary (Manning, Raghavan and Schütze, 2008).
- **Part of Speech tagging**: Part of speech tagging is the process of taking some text and breaking it down into their parts of speech. (NLP, no date)
- **Python Virtual Environment**: Python virtual environments are used to create self contained, isolated versions of python that allow you to install external python packages and modules without affecting other python installations. For example, *"If application A needs version 1.0 of a particular module but application B needs version 2.0, then the requirements are in conflict and installing either version 1.0 or 2.0 will leave one application unable to run."* Each application will have its own virtual environment and can exist simultaneously (Python Software Foundation, no datea).
- **Semantic Parsing**: *" Semantic parsing is the task of translating natural language utterances to (often machine executable) formal meaning representations"*
- **LSTM**: *"LSTMs are a way of storing information over extended time intervals by recurrent backpropogation."* Before, the introduction of LSTMs, recurrent backpropogation took long periods of time mostly because of insufficient, decaying error backflow (Hochreiter and Schmidhuber, 1997).
- **REST API**: Uses the HTTP protocol to transfer a resource and its state through HTTP protocols (Mozilla Developer Network, no date)
- **Docker Container**: *"A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another."* (Docker, no date)

Chapter 1

Introduction

1.1 Background

According to db-engines.com, 7 of the top 10 databases are relational databases (db-engines.com, no date). One method of extracting data from a relational database is the use of a language called SQL. Like all languages, SQL has certain syntax and rules that needs to be learned.

While quantifying the difficulty of learning the SQL syntax is difficult, English, the most common spoken language with "1.132 billion" speakers (Ethnologue, 2018), could potentially be used to make accessing and querying a database much easier. This lowers the barrier of entry on accessing relational data and can save time and resources by allowing non-technical users access to the database rather than relying on technical users to write SQL code for queries. This problem is part of a field called NLU.

1.2 Aims

The aims of this project is to take a query in english and convert it into the appropriate SQL query. However, to be able to show a practical use case, the process of converting will be done within the **context of retail**. For example, taking the phrase *"show me all the apples"* could be converted into the query below.

```
-- An example SQL query
SELECT * FROM products WHERE name='apple'
```

1.3 Formal Report

Since there are multiple ways to approach this problem, this project will consist of the following stages.

1.3.1 Data Gathering

Since the project will take part in a retail context, appropriate data needs to be gathered. The data that will need to be gathered will be **product data** . Product data such as product name, the type

of product as well as various other details will be gathered from the internet and will be discussed in more detail in Chapter 3.

1.3.2 Query Parsing

To convert the natural language queries to SQL. A variety of methods through a literature review will be gathered and discussed in more detail in the literature review in Chapter 2 and details of the actual implementation will be discussed in Chapter 4.

1.3.3 Testing parsing methods

To test the parsing methods, a sample mapping of natural language queries and expected SQL query will be made using the test dataset as discussed in Chapter 1.3.1 and Chapter 3. The parsing methods will be described in more detail in Chapter 4.

1.3.4 Evaluation

An evaluation of the parsing methods will then be written up taking into consideration of accuracy, measure of "closeness" to the correct solution and since the project is taking part in a retail context, challenges in deploying the methods into production.

Chapter 2

Literature review

There are currently two ways of approaching this problem. They are

1. **Analytical approaches**: This involves analysing the structure of the SQL database or schema as well as the natural language query in order to build a SQL query.
2. **Deep Learning Approaches**: This involves using complex machine learning models and neural networks trained on thousands of examples in order to build a complex model which when given a natural language query and other required inputs, can generate SQL queries

2.1 Analytical Approaches

2.1.1 FerreroJeremy/ln2sql

This GitHub repository (Ferrero, 2020) is based on the paper by (Couderc and Ferrero, 2015). While the original paper is in french, using the Google Chrome web browser will translate the paper to english although not perfectly.

2.1.1.1 Paper explanation

The overview for this paper is in Figure 2.1 which is extracted and translated from the French paper. The paper breaks down the model into the following steps. Since the paper was translated using Google Translate which is not 100% perfect, I have translated, rewritten and summarised the steps the model takes:

1. **Extraction of meaningful words**

This stage extracts meaningful words from the sentence entered by the user. TreeTagger is used filter out the **stop words**. Words that are kept include, but not limited to: common names, which can be the name of a table or column. Proper names, numbers, adjectives, etc can be thought to be column values

For example, the sentence:

> How old **are** the **students** whose **first name is Jean?**

The filter must return the elements: *"age, pupil, first name, Jean"*. The order must be preserved and is important for the next steps.

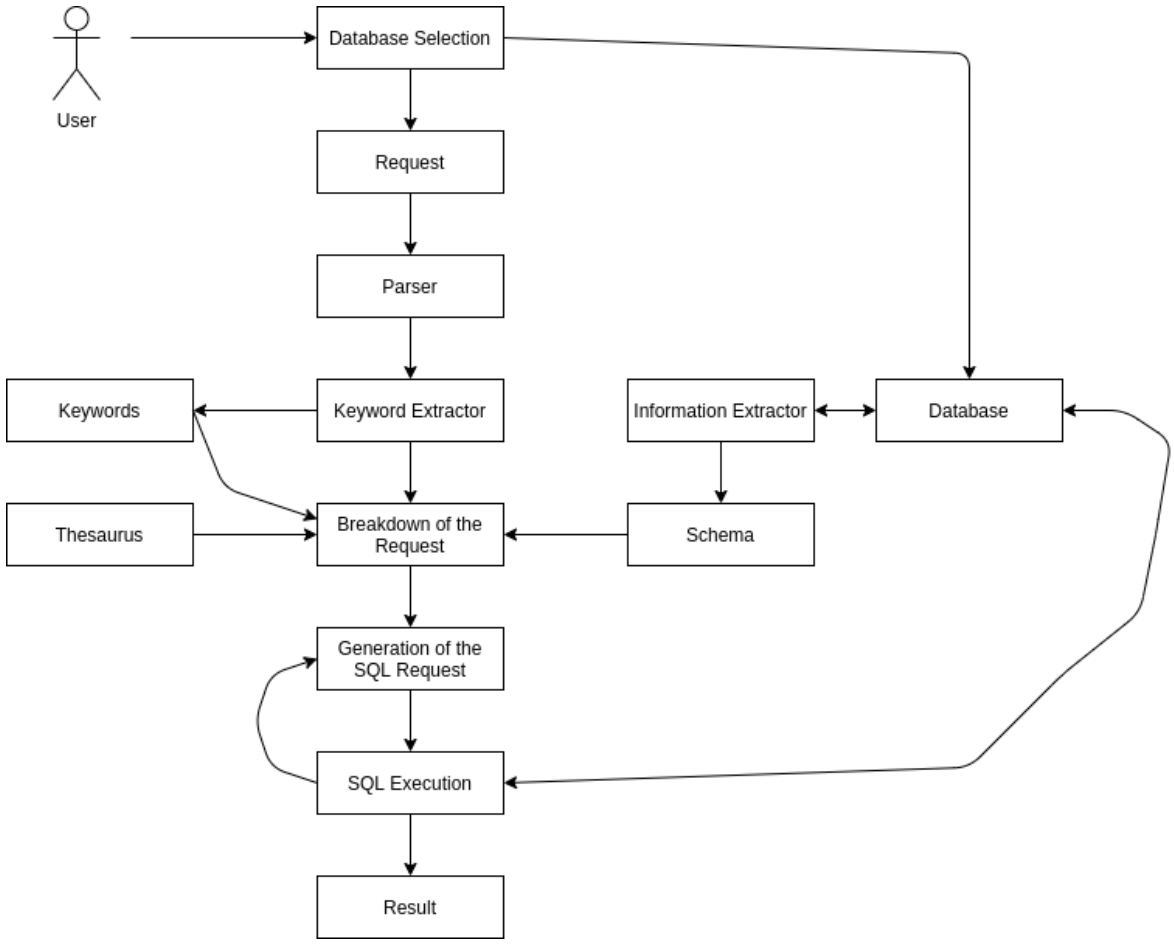

Figure 2.1: An overview of fr2sql taken from the fr2sql paper and translated from French to English

2. **Recovery of the architecture of the database**:

The second step of the process consists in recovering the architecture (structure) of the database on which the model is going to perform the queries on. To do this, the model needs to be aware of the database entities (columns, tables, primary and secondary keys, etc.) in order to allow a matching with the words extracted from the user request in Part 1.

Two methods have been implemented to achieve this result. The first method is to collect information necessary by querying the database using SQL queries of the type *"SHOW TABLES, SHOW COLUMNS,DESCRIBE, etc."*

The second method, consists in analyzing the backup or creation file for the database. By analyzing this file, a connection to the database is not required but a universal SQL schema is necessary (some commands being syntactically different under MySQL or Oracle for example). **Note**: fr2sql is therefore only compatible with an SQL database.

3. **Matching to a thesaurus**:

If the user does not enter a sentence written correctly or if the vocabulary is not **strictly** the same as that of the database, no match between the database and the sentence will be found and no relevant SQL query will be returned. Therefore a thesaurus will be created. For each word, there will be a table matching words to concepts together. For example, the words *"pupil"* and *"students"* represent the same **concept** where a concept is an idea that can be represented by a word or group of words. The purpose of this translator is to make the query of a database accessible to a person who does not know the structure or keywords (table and column names) and therefore are likely to use a synonym of a word used in the database than the word itself.

4. **Breakdown of the request**:

At this stage of the process, the application has a list of keywords from the request and a thesaurus of words. The idea now is to find a match between the request and the database using the thesaurus in order to better understand the structure of the query to generate. During the match all the words are put in lowercase and each keyword found is tagged according to whether it is a column or a table of the database or something else even if it is still unknown at the moment.

Breakdown of the sentence is carried out according to Figure 2.2 using the keywords tagged *"table"* and *"column"* found in the sentence to find out which segment of the sentence corresponds to which part of the SQL query to build. The presence of a *"SELECT"* and *"FROM"* segment is mandatory in the sentence. The first indicates which will be the type of selection and on which element(s) exactly, the second specifies in which table(s) to look for the selection item. The *JOIN* and *WHERE* segments are optional. The JOIN segment is used for explicit joins (Part 5) and the WHERE to specify, if any, constraints on the selection.

Depending on the number and the position of the key words in the sentence, output is not the same and therefore will not give the same output. **Note** that if a request contains no word that relates to a table, it will be invalid and will therefore generate an error.

5. **Determining the structure of the request**

For each of the segments obtained in Part 4, we analyze the keyword tagged *"unknown"*. These words could be algebraic calculations, counting query etc, or even a value which should be a constraint. That way if a word referring to a **count** such as *"how much"* is found in the first segment of the sentence, which corresponds to *SELECT*, the system identifies the request to be a count request, i.e. *SELECT COUNT (*)* which is first branch of the first segment in Figure 2.2. fr2sql uses a keyword recognition system in the SELECT and/or WHERE segments for many others types of operations. Table 2.1 is a non-exhaustive list of these keywords and their operations.

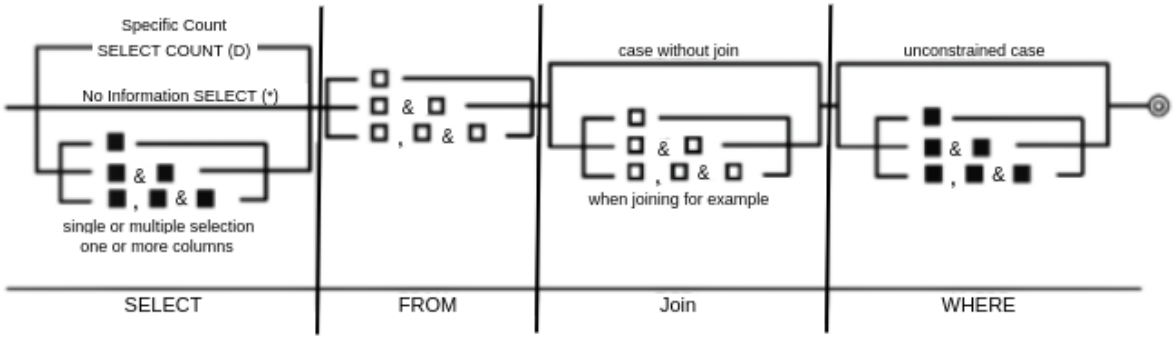

Figure 2.2: Diagram representing the division made on the sentence entered by the user in order to know what type of query to generate in output, the white squares representing the tables and the black squares the columns

Table 2.1: None exhaustive list of keywords and their operations

Keyword	operations
what is the number/how many are there/...	counting
Do not [...]/do not [...]/...	negation
Greater than/greater than/...	superiority
Less than/less than/...	inferiority
What is the sum/add/...	aggregate
What is the average/...	average

The paper also discusses **INNER JOINs**. Two types of inner joins exist, **implicit** and **explicit**. When there is a selection or constraint on a column that is not part of the *FROM* table, it is an **implicit join**. In this case it is necessary to make a join between the table of the target column and the table *FROM* which is specified in the sentence.

In the case of an explicit join, the table on which we must perform the join is specified directly in the sentence. Using Figure 2.3 as the database structure and the sentence:

> "Which students have a professor whose the first name is Jean?"

Here we must make a join between the table *eleve* (translates to student) and *professeur* (translates to professor) in order to be able to select all the students constrained on the teachers first names. The construction of the joins by the application is made possible by part 2 as fr2sql knows the primary and foreign keys of each table and can implicitly deduce the effective links between the tables and therefore knows if a table can be linked to another and if so, by which table(s) to pass. fr2sql can create joins through multiple tables if required (as in Figure 2.3, to access the teacher table from the table high through the table teaching and class).

If the selection or constraint column is neither in the *FROM* table, or in a table accessible by joining from the *FROM* table, then the request is impossible to build.

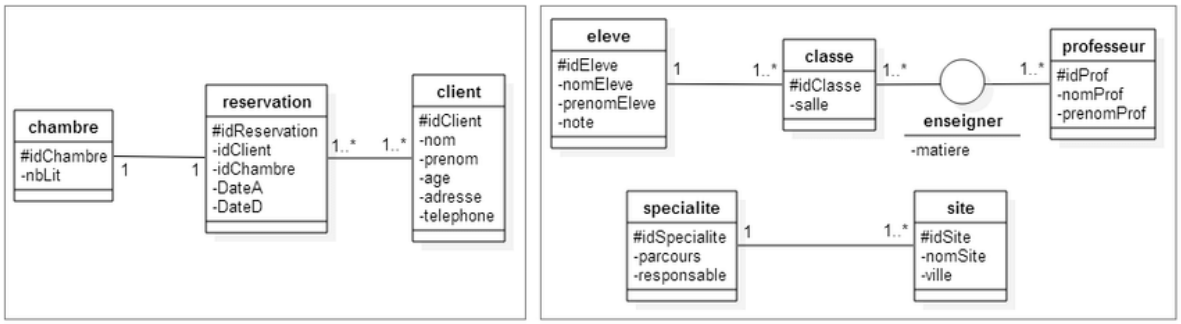

Figure 2.3: Entity Relationship diagram for explicit join. Taken directly from the paper. Some French-English translations: [enseigner = teach, matiere = Topic, eleve = student, professeur = professor]

Table 2.2: Some examples of equivalences between the keywords identified and the queries to be generated

Rules	Notable transactions
NB_KW + TABLE	selection with counting
TABLE	selection of all the columns of the table
TABLE + (TABLE) * + and + TABLE	the same request asked on several tables
COLUMN + TABLE	single column selection
COLUMN + (COLUMN) * + and + COLUMN + TABLE	multiple column selection
TABLE + COLUMN + VAL	constrained selection

6. **Generating the query using a *"lax"* grammar**

The *"permissiveness"* of existing techniques is due to their *"too tolerant"* matching. Given that there is a search for a finite set of data in a fairly large space, strict grammars are used to reduce the space of possible requests until proposing the most plausible solution. fr2sql uses Bidirectional matching to reduce the space for possible requests since it intersects a small dataset with another *"weak"* dataset.

A lax grammar is used to generate the output request adn the rules of this grammar is used to generate the query predetermined by matching. Although this technique adds significant processing time and preprocessing operations it significantly reduces the discrimination of matches and thus allows the use of a grammar which does not aim to have the most discriminating rules possible as the previous steps will have already filtered the majority of the sources of errors.

Now that we know the structure of the query to generate as output, we just need to replace the variable, table and column by their true value or name. Some examples of equivalences are available in Table 2.2

The request is therefore generated based on the presence, number and order of the keywords identified in the sentence input by the user. We call this grammar "lax" because it has enough rules to give the impression that the model accepts all forms of natural language queries.

Note that the problem of **mute constraints** is not supported by fr2sql. Requests such as "What are the students called Jean?"Or" Who are the 18 year old students? Will not be processed correctly by the application. Here,the column on which the constraint must be performed is implicit, it is not clearly specified, so it is impossible for the application to find it.

Results of Model

Below in Figure 2.4 are the results when tested against a manually written dataset against the database in Figure 2.3

	WORLD	SQL-HAL	English2SQL	fr2sql
Column selection	YES	YES	YES	YES
Selection on table	YES	YES	YES	YES
Multiple selection	NO	YES	YES	YES
Counting	YES	YES	YES	YES
Multiple counting	NO	YES	YES	YES
Simple constraint	YES	YES	YES	YES
Mute constraint	YES	NO	YES	NO
Disjunction	YES	YES	YES	YES
Conjunction	YES	YES	YES	YES
Cross stress	YES	YES	YES	YES
Date management	NO	YES	YES	NO
Storage	NO	NO	NO	YES
Comparison	YES	YES	YES	YES
Algebraic calculation	NO	YES	YES	YES
Negation	YES	NO	YES	YES
Synonymy	NO	NO	NO	YES
Join for selection	YES	NO	YES	YES
Join for condition	YES	NO	YES	YES
Nested queries	YES	NO	YES	NO
Compatibility across multiple databases	NO	YES	YES	YES
Vocabulary or grammar restriction	YES	YES	YES	NO

Figure 2.4: YES means operation is supported, NO means it is not

Also find below in 2.5 you can find results against differrent categories with different metrics where

$$precision = \frac{correct\_answers\_returned}{responses\_returned}$$

$$recall = \frac{correct\_answers\_returned}{entered\_sentences}$$

f-measure is *"can be interpreted as a weighted average of the precision and recall,"* (sklearn, no date)

	Precision	Recall	F-measure
All types of requests	0.939	0.850	0.892
Selections only	1	0.969	0.984
With joints	0.832	0.702	0.761
With conditions	0.987	0.880	0.930

Figure 2.5: Results against differrent categories with different metrics

2.1.1.2 GitHub Implementation

The GitHub repository is the implementation of the paper in Chapter 2.1.1.1. The implementation does not follow the paper exactly and deviates from the original in a few ways. These are

1. **Learning data model**

The original paper proposed both a database connection as mentioned in Chapter 2.1.1.1 part 2 by querying the database for information about the database and by analysing the database creation file (also knows as a dump file). This implementation only requires a dump file therefore no database connection is required.

2. **TreeTagger**

The original used TreeTagger for not only Part-of-speech tagging, but also to filter out stop words. This implementation uses personal configuration files for **languages**, **stop words** and **synonyms** in order to be more generic. One benefit of this is that all all languages can be supported as long as you have the appropriate configuration files. The repository has the configuration files for english built in.

3. **Output format**:

The original paper used a hash map for query structure generation (Chapter 2.1.1.1 part 6). This implementation uses a Python class in order to output the query in a JSON structure. The implementation also takes advantage of multi-threading for performance reasons.

The implementation also uses Figure 2.6 to display a simplified version of the overall architecture of the **program** where as the paper describes how each component works to build the query.

2.2 Deep Learning Approaches

2.2.1 Seq2SQL

Seq2SQL is a machine learning model that uses deep neural networks and policy based reinforcement learning to "translate natural language questions to corresponding SQL queries" (Zhong, Xiong and Socher, 2017). The model itself is not of interest but rather, a by product of this paper: **WikiSQL**

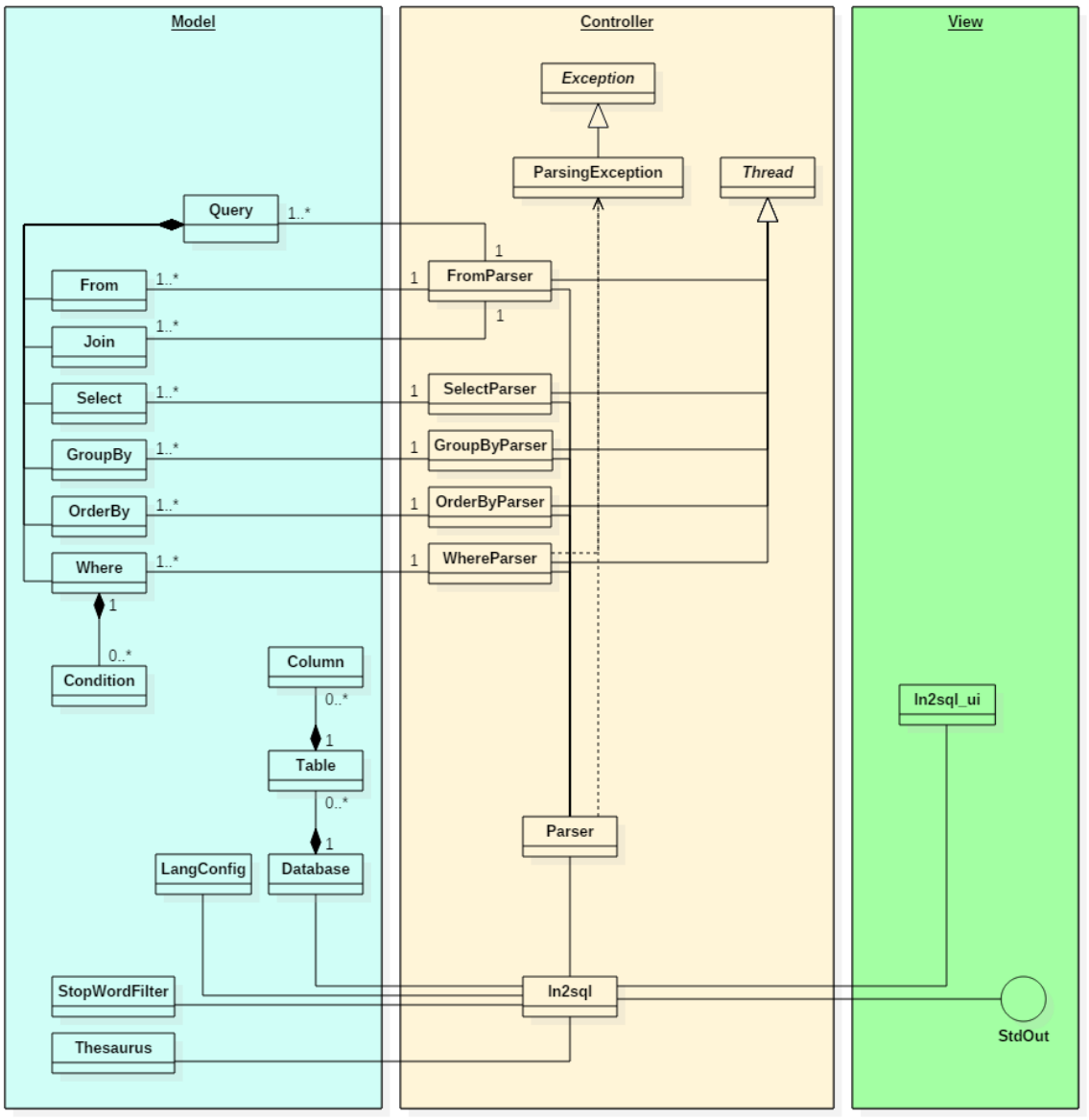

Figure 2.6: An overview of ln2sql taken from the GitHub repository

2.2.1.1 WikiSQL

The release of this paper also included the release of the WikiSQL dataset on GitHub (salesforce, 2019). This dataset includes *"a corpus of 80654 hand-annotated instances of natural language questions, SQL queries, and SQL tables extracted from 24241 HTML tables from Wikipedia."* This dataset, compared to related datasets, is the largest dataset in this specific field [(Zhong, Xiong and Socher, 2017), `Chapter 3 table 1`] and has also been used as a test dataset for other models details of which are available on the WikiSQL GitHub (salesforce, 2019)

2.2.2 paulfitz/mlsql

This GitHub repository (Fitzpatrick, 2020) is based on this GitHub repository (naver, 2020) which in turn is based on the paper by (Hwang *et al.*, 2019)

2.2.2.1 Paper Explanation

This paper attempts to perform semantic parsing on natural language in order to convert it into SQL queries. This paper uses the popular WikiSQL dataset mentioned in Chapter 2.2.1.1. Below is a brief summary of how the paper by (Hwang *et al.*, 2019) works.

2.2.2.1.1 Model

1. **Table-aware BERT encoder**

Bidirectional **E**ncoder **R**epresentations from **T**ransformers (BERT),

> *"is designed to pretrain deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers. As a result, the pre-trained BERT model can be fine tuned with just one additional output layer to create state-of-the-art models for a wide range of tasks, such as question answering and language inference, without substantial task specific architecture modifications."* (Devlin *et al.*, 2019)

In SQLova, BERT is used to encode the natural language query in combination with SQL table headers. `[SEP]` (a special token in BERTS) is used to separate between the query and the headers. For each query input $T_{n,1}, ..., T_{n,1}$ where L is the number of query words, the natural language query is encoded according to Figure 2.7

Another input to BERT is the segment id, which is either 0 or 1. We use 0 for the question tokens and 1 for the header tokens. Other configurations largely follow (Devlin *et al.*, 2019). The output from BERT are concatenated and used in **NL2SQL**

$$[\mathrm{CLS}],\ T_{n,1},\ \cdots\ T_{n,L},\ [\mathrm{SEP}],\ T_{h_1,1},\ T_{h_1,2},\ \cdots,\ [\mathrm{SEP}],\ \cdots,\ [\mathrm{SEP}],\ T_{h_{N_h},1},\ \cdots,\ T_{h_{N_h},M_{N_h}},[\mathrm{SEP}]$$

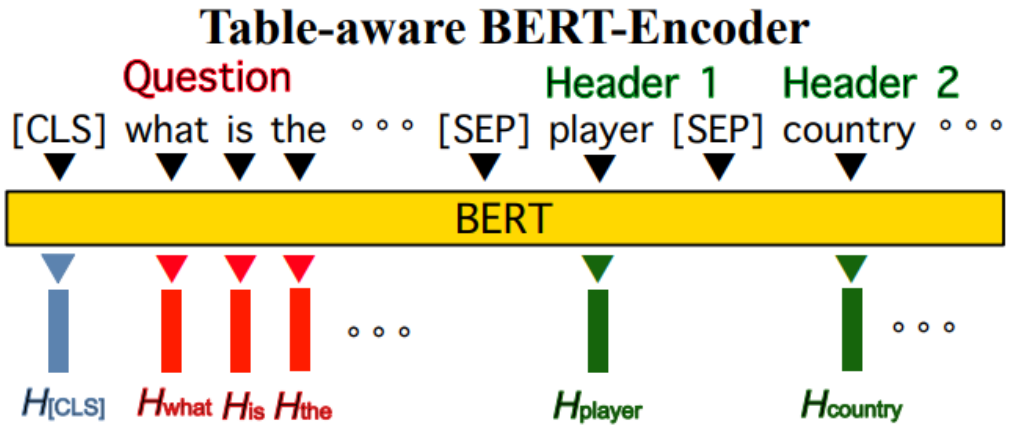

Figure 2.7: The scheme of input encoding process by table-aware BERT. Final output vectors are represented by colored bars: light blue for '[CLS]' output, red for question words, and green for tokens from table headers.

2. **NL2SQL**

This layer is described in Figure 2.8 on top of BERT.

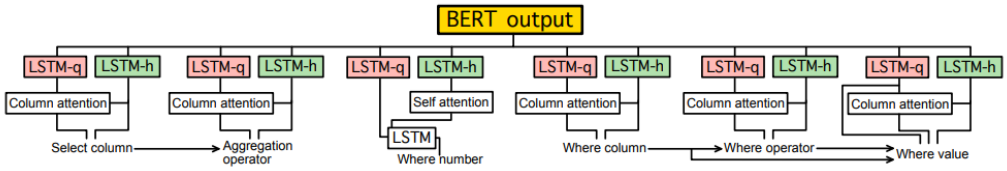

Figure 2.8: The illustration of NL2SQL LAYER. The outputs from table-aware encoding layer (BERT) are encoded again with 'LSTM-q' (question encoder) and 'LSTM-h' (header encoder).

NL2SQL Layer uses syntax-guided sketch, where the generation model consists of six modules, namely `select-column`, `select-aggregation`, `where-number`, `where-column`, `where-operator`, and `where-value`. Also, following, column-attention is frequently used to contextualize question.

- `select-column`: finds the `select` column from natural language
- `select-aggregation`: finds aggregation operator `agg` for given column c among six possible choices (`NONE`, `MAX`, `MIN`, `COUNT`, `SUM`, and `AVG`)
- `where-number`: finds the number of `where` condition(s)
- `where-column`: obtains the probability of observing a particular column
- `where-operator`: finds `where` operator `op` where $op \in =, >, <$ for given column c
- `where-value` finds `where` condition by locating start-token and end-tokens from question for given column c and operator op.

3. **Execution-Guided Decoding (EG)**

During the decoding (SQL query generation) stage, non-executable (partial) SQL queries can be excluded from the output candidates for more accurate results.

In `select` clause, (`select` column, aggregation operator) pairs are excluded when the string-type columns are paired with numerical aggregation operators such as `MAX`, `MIN`, `SUM`, or `AVG`. The pair with highest joint probability is selected from remaining pairs.

In `where` clause decoding, the executability of each (`where` column, operator, value) pair is tested by checking the answer returned by the partial SQL query $selectagg(col_s)wherecol_wopval$. Here, col_s is the predicted `select` column, `agg` is the predicted aggregation operator, col_w is one of the `where` column candidates, `op` is `where` operator, and `val` stands for the `where` condition value. The queries with empty returns are also excluded from the candidates. The final output of where clause is determined by selecting the output maximizing the joint probability estimated from the output of `where-number`, `where-column`, `where-operator`, and `where-value` modules.

Results

According to the GitHub repository for sqlova, the results for the model with and without Execution Guided Decoding is as follows

Model	Dev logical form accuracy	Devexecution accuracy	Test logical form accuracy	Test execution accuracy
SQLova	81.6 (+**5.5**)^	87.2 (+**3.2**)^	80.7 (+**5.3**)^	86.2 (+**2.5**)^
SQLova-EG	84.2 (+**8.2**)*	90.2 (+**3.0**)*	83.6(+**8.2**)*	89.6 (+**2.5**)*

- ^: Compared to current SOTA models that do not use execution guided decoding.
- *: Compared to current SOTA.
- The order of where conditions is ignored in measuring logical form accuracy in our model.

2.2.2.2 GitHub Implementation

(Fitzpatrick, 2020) takes **SQLova** which is based on this GitHub repository (naver, 2020) and puts it inside a Docker container. There are then instructions to download and setup the container. Once the container is running, a REST API is exposed for you to send a request which contains:

1. A plain text question
2. A `csv` file which contains an SQL table

The container then return a JSON response

```
{
    "answer": [
        41908
    ],
    "params": [
        "large bottled drinks",
        "1"
    ],
    "sql": "SELECT sum(id) FROM products WHERE category = ? AND price > ?"
}
```

Chapter 3

Data Gathering

As mentioned in Chapter 1.3.1, a test dataset is required in order to evaluate the different parsing methods as discussed in Chapter 2.

3.1 Gathering the test Data

As discussed is Chapter 1.3.1, the data we will be gathering is **product data**. In order to obtain this data, we will be web scraping. The website we will be using is https://www.bestwaywholesale.co.uk/ (Bestway Wholesale, no date) and we will use **scrapy**, a python library, to perform the web scraping (scrapy, 2019) and BeautifulSoup4, another python library, (Richardson, no date) for parsing the HTML.

3.1.1 Scraping

As mentioned in Chapter 3.1, we use scrapy, a python package to iterate through webpages, extracting the information required and storing it in a database. The database we are using is SQLite (SQLite Team, no date) and SQLAlchemy (sqlalchemy, 2019) is used as the ORM (see Chapter 0.3 for more information on ORMs). A file called `requirements-dataGathering.txt` contains a list of all the python packages used for data scraping

Using (Krebs, no date) as guidance on how to use SQLAlchemy, A database schema was designed and is visualised in figure 3.1 as an ERD.

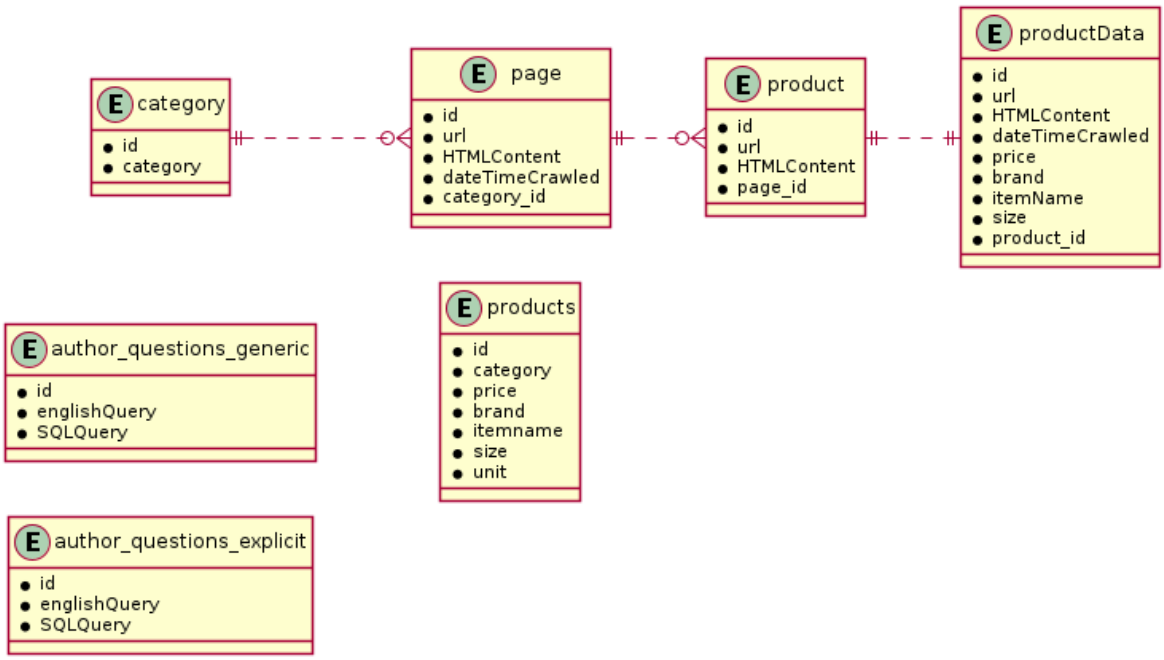

Figure 3.1: Entity Relationship Diagram for the dataset

The entities mentioned in figure 3.1 are explained in Appendix A.

> **Note**: Not all the entities in the diagram above are used in the dataset scraping. The entities used in the dataset scraping are found in Appendix A.1.

3.1.2 Scraping Process

The scraping process involves various files. Below is a description of each file used in the scraping process and their function. The files are given as a path from the root of the GitHub repository that the files are stored in (Patel, 2019) (Note that the files below are **directly** involved in the scraping process).

- `table_definitions/base.py`: Used for defining the database.
  - **Note**: This file contains a hard coded path to the dataset (the file in question is `dataset.sqlite3`) and will have to be modified before interacting with the database using SQLAlchemy. More details available in the repository `README.md` (Patel, 2019)
- `table_definitions/category.py`: A class used to create rows in the category table.
- `table_definitions/page.py`: A class used to create rows in the page table.
- `table_definitions/product.py`: A class used to create rows in the product table.
- `table_definitions/productdata.py`: A class used to create rows in the productdata table.

- `html_parse/page_product.py`: A function which given a category, gets the associated page rows. For each HTMLcontent in page row, find all the products elements and store in the database.
- `html_parse/productdata.py`: A function which converts a HTML table to a dictionary. Used to extract detailed information about a scraped product. Used by `scrapers.py` (specifically `class crawlProduct(Spider)`).
- `database.py`: Used to create the actual database file. Uses `table_definitions/base.py` to help create the database.
- `scrapers.py`: The code responsible for visiting the web pages and getting the HTML that is going to be parsed.
- `raw_data_pipeline.py`: The "glue" code that takes all the files mentioned above and performs a complete scrape from a category to scraped products.

The scraped data is available in the file **dataset.sqlite3** in this reports associated GitHub repository (Patel, 2019).

Note: Some of the data has been manually cleaned. The manual cleaning is included, but not limited to

1. Manually inserting values where the values were null
2. Making the contents of the database all lowercase

The final dataset that will be used to test the data is described in more detail in Appendix A.2

3.1.3 Other files

- `size_split/split_size.py`: used to separate the size of the products into the size and its unit (e.g. 1.5kg is transformed into 1.5 and kg as two separate values). The script takes the products table as a csv file, split the size and unit into their own table and save the output as a csv, this is then imported into the SQLite database.
- `lowercase_products.py`: simple script to take each row in the `products` table and make all the letters lowercase
- `utils.py`: Miscellaneous utility functions such as formatting text in a terminal window

3.2 Gathering Natural Language Queries

In order to test the various methods, a set of natural language queries as well as their corresponding SQL queries will be needed. The questions will be stored in `dataset.sqlite3` in the tables `author_question_explicit` and `author_question_generic` details of the SQL table can be found in Appendix A.3

Table 3.1: Sample explicit and generic questions

Explicit	Generic	SQL query
show me all the products where the category is large bottled drinks	show me all the products in the large bottled drinks category	SELECT * FROM products WHERE category = large bottled drinks

3.2.1 Natural Language Queries

One of the aims of the reports is to find out is how different approaches compare to each other. One way to test this is to have a NLQ which is then mapped to an SQL query.

In order to be more thorough in testing, there will be two sets of questions

1. author_question_explicit
2. author_questions_generic

The explicit dataset will have their natural language queries quite *"rigid"*. What this means that the natural language queries will explicitly mention the names of the columns and will attempt to write the natural language queries similar to how the equivalent SQL query will be written while trying to avoid using words found in the SQL syntax and being written to sound like a natural language query. The generic dataset will have the exact same SQL queries but will attempt to have natural language queries that sound more natural and less rigid. This will attempt to more closely mimic natural speech.

The reason for doing this is to test the robustness of the models as well as attempting to *simulate* how these models will be used in a production environment as a service like this is likely to be used by different people and different people phrase things differently but yet can mean the same thing when mapped to SQL.

An example from the explicit and generic questions is shown in table 3.1 to better illustrate the differences between the questions. Note that explicit and generic both map to the same SQL query but explicits sentence structure is more similar to the SQL query than generic.

Chapter 4

Query Parsing

This chapter will deal with taking the natural language queries, feeding them into the parsing methods and collecting their results. It will also discuss any changes made to the code as well as well as how to execute the programs.

4.1 FerreroJeremy/ln2sql (fork)

The repository was forked and some modification were made. You can find the forked repository here (Patel, 2020a) or https://github.com/zayd62/ln2sql. The changes are described under **Modified/Additional Files**.

4.1.1 Modified/Additional Files

4.1.1.1 ln2sql/demo/CLITable.py

This file was made by (Goldsborough, 2015). Its purpose is to take a nested list and output a table in the terminal with the appropriate layout and formatting

```
data = [["col1", "col2"],["row1,col1", "row1,col2"],["row2,col1", "row2,col2"]]
```

The output of the file is as follows;

```
+-------------------------------------+-------------------------------------+
|col1                                 |col2                                 |
+-------------------------------------+-------------------------------------+
|row1,col1                            |row1,col2                            |
+-------------------------------------+-------------------------------------+
|row2,col1                            |row2,col2                            |
+-------------------------------------+-------------------------------------+
```

4.1.1.2 ln2sql/demo/utils.py

This was made by (Hadida and sheljohn, 2018). Its purpose is to take a string and convert it so that when text is rendered on a terminal or shell, the text is rendered with certain colours and styles (e.g. **bold**, underline, *italic*, etc)

4.1.1.3 ln2sql/demo/interactive.py

This file is an interactive demo of parsing natural language queries to SQL queries. It takes a natural language query as input and return the generated SQL query. It also executes the query against the test dataset described in more detail in Chapter 3

> **Note**: There are hard coded paths in this file. Consult `README.md` on what to change in order for the file to work.

4.1.1.4 ln2sql/demo/test_dataset_explicit.py and ln2sql/demo/test_dataset_generic.py

both of these files are responsible for executing the test for the explicit dataset and the generic dataset respectively.

> **Note**: There are hard coded paths in this file. Consult `README.md` on what to change in order for the file to work.

4.1.2 Using interactive demo

In order to run the interactive demo, it is recommended to create and then activate a Python Virtual Environment (tested on Python 3.7.7).

> **Note**: if you don't have Python 3.7.7 available on your system, you can use Conda to not only manage Virtual Environments, but also specify which Python version to install. Instructions for installing specific Pythons are available here

Then download the repository linked in Chapter 4.1 and before running, there are some hard coded file paths that need to be changed, details of which are available in the file `README.md`. After adjusting the variables, open a terminal in the root of the downloaded repository and run the following command to launch the interactive demo of converting language queries into SQL.

```
python -m ln2sql.demo.interactive
```

4.1.3 Running against the test dataset

There are two test datasets as discussed in Chapter 3.2.1. running the following commands bellow will run the tests against both datasets. each command will also print a file path to where the results files are saved to.

```
python -m ln2sql.demo.test_dataset_explicit
python -m ln2sql.demo.test_dataset_generic
```

For explicit, the test results will be in a file called `test_results_ln2sql_explicit.csv` and for generic, `test_results_ln2sql_generic.csv`

4.2 paulfitz/mlsql implementation

4.2.1 Prerequisites

Visit the implementation repository here (Patel, 2020b) and follow the instructions on how to setup the implementation.

4.2.1.1 Same Files

The following files are exactly the same as in Chapter 4.1

- CLITable.py
- utils.py

4.2.1.2 New file

- **products.csv**: this is an SQL dump of the database. This is used for returning answers to natural language query as well as for the model to learn the database structure.

4.2.1.3 Running interactive and tests

Since the setup of this method is much more complicated, detailed instructions are in the repositories `README.md` file available here (Patel, 2020b) as well as instructions on how to run the demonstrations.

Chapter 5

Evaluating Parsing Methods

Since the project is taking part in a retail context as mentioned in Chapter 1.2 and that different techniques will be compared as mentioned in Chapter 1.3.2, being able to evaluate each technique is important. This chapter discusses the evaluation metrics that will be used in order to determine the better parsing method as well as the results of feeding the test dataset into the parsing methods.

5.1 Evaluation Metrics

5.1.1 Accuracy

This metric is simply a measure of how many SQL queries did the methods generate that match the actual solution. This metric however does have some issues of which will be discussed below.

5.1.1.1 Results Equivalency

According to the ISO/IEC 9075:1992 which is the 1992 standard of SQL (International Standards Organisation, 1992) , under Chapter 13: **Data Manipulation** sub-section **<declare cursor>** it states under **General Rules**

> 2) If an <order by clause> is not specified, then the table specified by the <cursor specification> is T and the ordering of rows in T is implementation-dependent.

Since the queries above do not have an `ORDER BY` clause, the ordering of the rows in table 5.2 is not guaranteed to be the same across different implementations of the SQL language by different relational databases (MySQL, PostgreSQL, SQLite are all examples of relational databases that implement SQL). For example, MySQL might return the items in table 5.2 with `id=0` first then `id=1` where as SQLite might return `id=1` first then `id=0`.

5.1.1.1.1 Solution The solution to this is to ensure that each entry in the SQL table has a numerical unique identifier that way, the generated query and its results can be further sorted by its identifier in order to ensure that results of the query are equivalent regardless of database as the queries can be sorted and compared by its identifier.

Another solution is to run all the parsing methods in a single relational database that way, the ordering of the rows are consistent as row ordering is implementation specific. Since SQLite is already being

Table 5.1: An example SQL table called "test"

id	name	size	category	brand
0	coca-cola	330ml	drinks	Coke
1	diet coke	330ml	drinks	Coke
2	plain flour	1000g	baking	tesco
3	white medium bread	500g	bakery	hovis

used to store data related to the project (see Appendix A) SQLite is the database that all the parsing methods will use.

SQLAlchemy, according to it documentation, uses `pysqlite` which *"is the same driver as the sqlite3 module included with the Python distribution."* In the interest of reproducibility, the sqlite3 driver version is described bellow as well as the python code used to obtain the version:

```
import sqlite3
print(sqlite3.sqlite_version)
```

```
## 3.22.0
```

5.1.1.2 SQL Query Equivalency

There are some **SQL queries** that are syntactically different but when executed, return the same result. Table 5.1 will be used to demonstrate this issue.

If we want to, for example, run a query that will *"select all the products that are in the category drinks"*, then the following SQL queries will return the same results. Each query also has a short description on what differentiates it from the other queries and is not an exhaustive list of all possible combinations.

```
-- Query 1: Common SQL query
SELECT * FROM test WHERE category='drinks'
-- Query 2: Selecting table columns using table_name.column_name
SELECT test.id, test.product_name, test.size, test.category, test.brand FROM
    test WHERE category='drinks'
-- Query 3: Selecting table columns using column_name only
SELECT id, product_name, size, category, brand FROM test WHERE category='drinks'
-- Query 4: Mixed use of lowercase and UPPERCASE SQL keywords
select id, product_name, size, category, brand FROM test
    where test.category='drinks'
```

The results of all 4 queries above is as follows

Table 5.2: Results of the query

id	name	size	category	brand
0	coca-cola	330ml	drinks	Coke
1	diet coke	330ml	drinks	Coke

5.1.1.2.1 Solution The solution is to determine if the queries are equivalent or not by looking at the results of the query. If they are equivalent, then they will return the same results. One way to do this is to execute both the author SQL query and the generated SQL query and see if the results match, if they do, then we can consider the two queries as equal and that the model has correctly generated the SQL query.

5.1.2 Ease of use

Since the end user will receive the SQL query and (possibly), the results of the query executed, this section will discuss how *"easy"* it is for IT Infrastructure staff and/or Application developers to use the query parsing methods. Since this *"easy"* is not quantifiable, the following observations will be made and will be left up to the reader to decide. The observations are:

1. How easy it is to install
2. Resource requirements
3. Using the parsing method

5.2 FerreroJeremy/ln2sql Evaluation

5.2.1 Accuracy

When ran against the test dataset as mentioned in Chapter 3.2, we measured how many queries the method calculated (using the method described in Chapter 5.1.1.2.1) correctly and how many incorrectly across both generic and explicit. The results are available in Figure 5.1

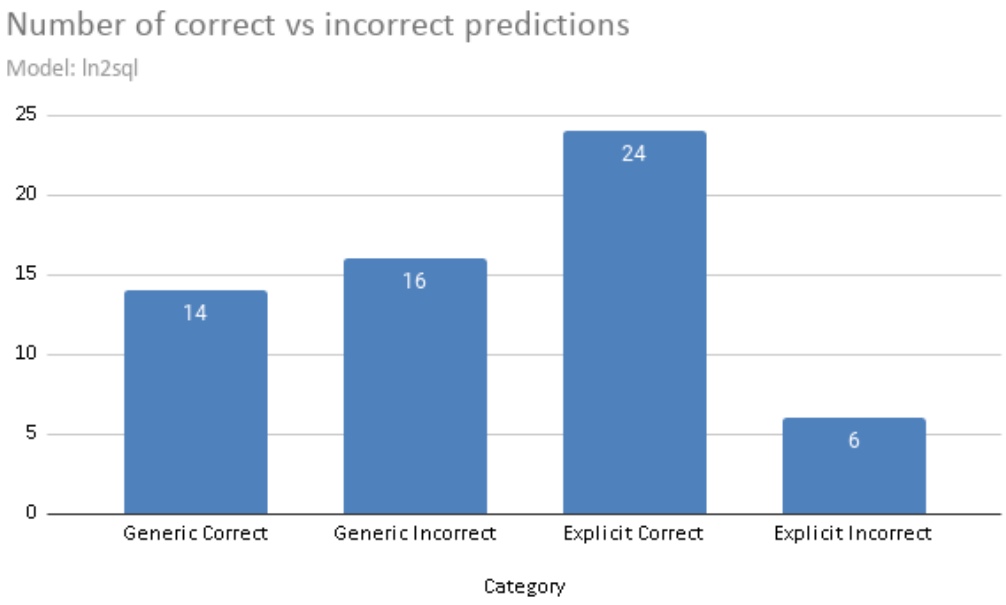

Figure 5.1: results for ln2sql

Table 5.3: queries of interest in explicit

question id	author questions	author sql	generated sql	correct prediction
4	show me all the products where the category is ”biscuits” or the category is ”hot drinks”	SELECT * FROM products WHERE category = ”biscuits” OR category = ”hot drinks”	SELECT * FROM products WHERE products.category = ’biscuits’ AND products.category = ’hot drinks’	False
8	show me all the products that has a price less than 1 or a price greater than 2	SELECT * FROM products WHERE price < 1 OR price > 2	SELECT * FROM products WHERE products.price < ’1’ OR products.price > ’2’	True
20	show me the price and itemname of all the products in the crisps category	SELECT price, itemname FROM products WHERE category = ”Crisps”	SELECT products.price, products.itemname FROM products WHERE products.category = OOV	False

The graph on its own suggest that a natural language query that is more explicit is more likely to generate a query that returns the same result as the author query but not perfect whereas a generic query has approximately a 50% chance of calculating it correctly.

Looking at the results in `Rmarkdown Report/results/ln2sql/`, for explicit it managed to correctly generate 80% of the SQL queries. One particularly interesting query it got wrong was the `id = 4` in Table 5.3. Even though the word *“or”* was explicitly mentioned, ln2sql generated an *“and”*. What is also interesting is that `id = 8` was able to correctly generate an *“or”* although the fact that a different column and different comparator was used. Running, (in interactive mode), `show me all the products where the price is 2 or the price is 1` gives `SELECT * FROM products WHERE products.price = '2' OR products.price = '1'` so given that the comparator is the same, we could infer that it is either the column that causes this deviation, or the fact that we are comapring intergers rather than strings which leads to the model suggesting an *“or”*.

Table 5.4: queries of interest in generic

question id	author questions	author sql	generated sql	correct prediction
1	show me all the products in the large bottled drinks category	SELECT * FROM products WHERE category = "large bottled drinks"	SELECT * FROM products WHERE products.category = OOV	False
2	show me all the products in the crisps category	SELECT * FROM products WHERE category = "crisps"	SELECT * FROM products WHERE products.category = OOV	False
3	show me all the products in the hot drinks category	SELECT * FROM products WHERE category = "hot drinks"	SELECT * FROM products WHERE products.category = OOV	False
9	show me the brand and itemname from products	SELECT brand, itemname FROM products	SELECT products.brand, products.itemname FROM products	True
11	show me the maximum price of products	SELECT MAX(price) FROM products	SELECT MAX(products.price) FROM products	True
12	show me the minimum price of products	SELECT MIN(price) FROM products	SELECT MIN(products.price) FROM products	True
13	show me the average price of the products	SELECT AVG(price) FROM products	SELECT AVG(products.price) FROM products	True

For generic, the model performed much more poorly (Table 5.4). It got more than half wrong. `id 1 - 3` suggests that, in comparison to the explicit dataset, having the value first and column name second will result in an incorrect prediction where as having it the other way around (as in the explicit dataset) will result in the correct prediction. This is where the model mainly fails in the generic dataset. The model can handle simple queries such `id = 9 ,11 - 13` but apart from that, ln2sql is more suited to queries that follow the structure of a SQL query.

5.2.2 Ease of Use

1. **How easy is it to install**

Installing ln2sql is very easy as the repository has very easy to follow instructions and has most of the resources needed in the repository. While the original (Ferrero, 2020) has a very simple user interface as well as an Command Line interface (CLI) through the terminal, it is very easy use the same code that powers the CLI in your own code. This paper, for example used the code to make a terminal interface with colours as well as the ability to execute and return the queries. The code for that is available here (Patel, 2020a)

2. **Resource requirements**

ln2sql has no external dependencies and a significant slow down was not experienced. The only externel dependency was **pytest** a python testing framework ('Pytest-dev/pytest', 2020) so as long as you are able to run python (tested on Python 3.7), you should be able to clone the repository and start parsing natural language queries

3. **Using the parsing method**

The parsing method is just a simple as having an SQL dump file which contains the structure of the databse as well as a language configuration file (included with the repository) and feeding the program an english natural language. Keyword filtering can be improved through the use of a thesaurus (included) as well as the ability to improve the stop word filtering as well. A CLI tool and a GUI tool is included as well.

Conclusion

Overall, ln2sql is a very quick and easy way to start parsing SQL queries as it is simple to install, run wherever python can run and is open to improvement due to available source code, paper and various configuration files all available to tweak. One small caveat is that the built in GUI and CLI requires the `-m` flag (`python -m ...`) which according to the python CLI docs (accessed via `python -h` on Ubuntu 18.04 terminal), run library module as a script so you would probably have to convert the module to a python package which is beyond the scope of this paper.

5.3 paulfitz/mlsql

5.3.1 Accuracy

Just like Chapter 5.2.1, we measured the accuracy between the explicit and generic dataset and measured how many were correct or incorrect. The results are in 5.2

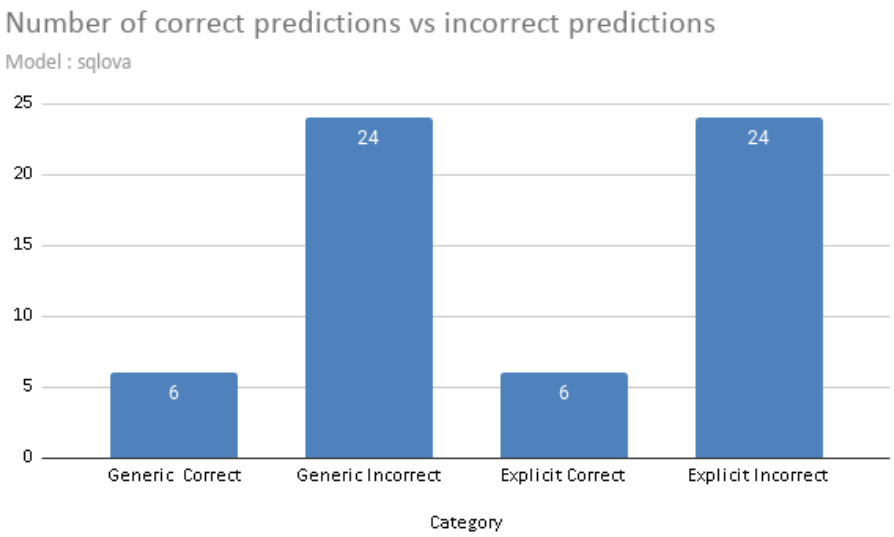

Figure 5.2: Results for mlsql

Table 5.5: first 5 tests in generic

question id	author questions	author sql	generated sql	correct prediction
1	show me all the products in the large bottled drinks category	SELECT * FROM products WHERE category = "large bottled drinks"	SELECT (itemname) FROM products WHERE category = 'large bottled drinks'	False
2	show me all the products in the crisps category	SELECT * FROM products WHERE category = "crisps"	SELECT (itemname) FROM products WHERE category = 'crisps'	False
3	show me all the products in the hot drinks category	SELECT * FROM products WHERE category = "hot drinks"	SELECT (itemname) FROM products WHERE category = 'hot drinks'	False
4	show me all the products in the biscuits or the hot drinks category	SELECT * FROM products WHERE category = "biscuits" OR category = "hot drinks"	SELECT (brand) FROM products WHERE category = 'biscuits or the hot drinks'	False
5	show me all the products where the price is less than 1	SELECT * FROM products WHERE price < 1	SELECT (itemname) FROM products WHERE price < '1'	False

Data suggests that sqlova performed worse than ln2sql based on the graph alone. However, taking a look at the data itself for generic Table 5.5, `id 1 to 3` appear to have the *where* filter correct, the *select* filter is *itemname* rather than *"*"*. Another reason is that sqlova is based on a deep learning approach so to get the best results, the model will need to be trained on some of samples of data relevant to the retail domain where as sqlova was trained on Chapter 2.2.1.1.

5.3.2 Ease of Use

1. **How easy is it to install**

Installation was very simple as the model was made available via a Docker Container so once the contaier was downloaded, only once command is need to run the models. One small issue is that the download was over 4GB so for slow internet conections, unreliable connections and meterred connections may not be able to use this.

2. **Resource requirements**

One benefit of Docker Containers is that no dependency management is needed but the Docker Software

itself is needed to run the container. A strong internet connection is needed to download the model as it is over 4GB and it requires 3GB of RAM which is quite when ran on consumer hardware.

3. **Using the parsing method**

The model is ONLY interacted with through a REST API so can be accessed programmatically. One way that this can be used by end users is by placing a user interface, e.g. Desktop of Web Application which then communicates with the API on the users behalf. Also, given the fact that this is a Deep learning approach, debugging and improving the will require specialist talent. Deploying the container designed to be simple due to the containarised nature.

Conclusion

While the REST API is nice from a developers perspective, given the weaker performance and higher memory usage, this approach is not as nad as ln2sql.

Chapter 6

Conclusion

In conclusion, we have taken two different approaches to the same problem, an analytical and deep learning approach. We have experimented and measured how they perform in a domain specific environment. While deep learning approaches may be able to give good or maybe better than human accuracy, given the specific domain (Retail), the deep learning approach may fail as it works on what it has learnt from thousands on training sessions and may not have any experience with this type of data. An analytical approach like ln2sql attempt to calculate the generated SQL query by breaking down the sentence into its individual components and rebuilding a query from those parts as well as an awareness of the database. In this case, an analytical approach had achieved a much higher success rate than the large and resource intensive Deep Learning model.

6.1 Further work

Further work could include:

- **More data**: Retail does not just consist of products, there is also the human element such as stores storing data about you for their rewards program so queries into a database with people data.
  - An extension to this could be different data types such as dates.
- **Training**: Using larger amounts of domain specific data, try and train the machine learning model and see if better performance can be obtained.
- **ln2sql options**: all the files such as the language configuration files were left as is. Perhaps experimenting with these file could yield a better result.

Appendix A

Dataset Schema Explained

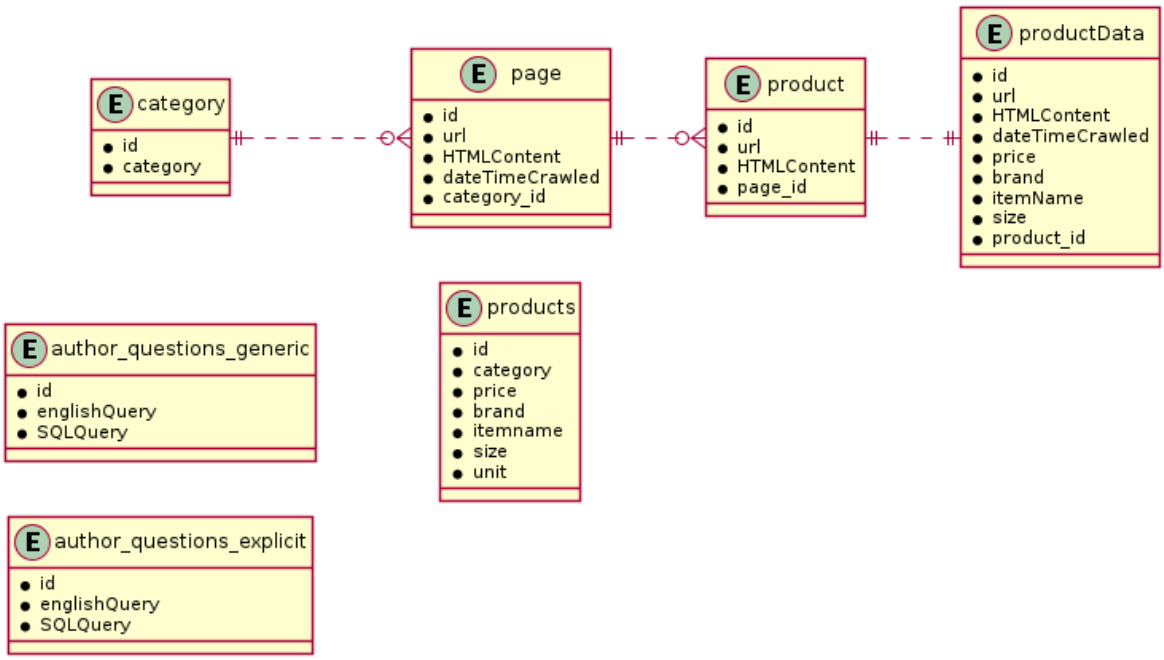

Figure 3.1 is repeated here. The dataset is stored in a file `dataset.sqlite3`

A.1 Raw Scrapped Dataset

Below are the tables used in scraping the raw data. Note that this is an internal table and will **NOT** be accessible to the parsers.

- **category**
  - **id**: used as database primary key
  - **category**: name of the product category
- **page**
  - **id**: used as database primary key
  - **url**: for each category, the product information may span several "pages". This field represents the url of each page
  - **HTMLContent**: stores the HTML of the url above
  - **dateTimeCrawled**: the date and time the url was crawled
  - **category_id**: used to identify which entry in the category table this page belongs to
- **product**
  - **id**: used as database primary key
  - **url**: for each page, the there are several product previews. This store the URL to find more detailed information about products.
  - **HTMLContent**: stores the HTML of the product preview. A page contains multiple product previews.
  - **page_id**: used to identify which entry in the page table this product belongs to
- **productData**
  - **id**: used as database primary key
  - **url**: Stores the url of the product we are crawling.
  - **HTMLContent**: stores the HTML of the url above
  - **dateTimeCrawled**: the date and time the url was crawled
  - **price**: price of the product
  - **brand**: the brand of the product
  - **itemName**: the name of the product
  - **size**: the size of the product
  - **product_id**: used to identify which entry in the product table this entry belongs to

A.2 Products table

The products table is the final table to be used as the test dataset. Note that originally, the size and unit columns where

- **products**
  - **id**: used as database primary key
  - **category**: name of the product category
  - **price**: price of the product
  - **brand**: the brand of the product
  - **item name**: the name of the product
  - **size**: the size of the product
  - **unit**: the unit of the size (e.g. ml, kg etc)

A.3 Question tables

The purpose of theses tables are discussed in Chapter 3.2.1

- **author_questions_explicit**
  - **id**: used as database primary key
  - **englishQuery**: The natural language query
  - **SQLQuery**: The corresponding SQL query that the model should generate
- **author_questions_generic**
  - **id**: used as database primary key
  - **englishQuery**: The natural language query
  - **SQLQuery**: The corresponding SQL query that the model should generate

References

Bestway Wholesale (no date) 'UK Cash & Carry Food, Drink, Foodservice & Pet Bestway Wholesale'. Available at: https://www.bestwaywholesale.co.uk/ (Accessed: 9 December 2019).

Chen, P. P.-S. (1976) 'The entity-relationship model—toward a unified view of data'. Association for Computing Machinery. Available at: https://doi.org/10.1145/320434.320440 (Accessed: 26 March 2020).

Couderc, B. and Ferrero, J. (2015) 'Fr2sql: Querying databases in French', in. Available at: https://www.researchgate.net/publication/280700277_fr2sql_Interrogation_de_bases_de_donnees_en_francais.

db-engines.com (no date) 'DB-Engines Ranking', *DB-Engines*. Available at: https://db-engines.com/en/ranking (Accessed: 14 March 2020).

Devlin, J. *et al.* (2019) 'BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding', *arXiv:1810.04805 [cs]*. Available at: http://arxiv.org/abs/1810.04805 (Accessed: 12 May 2020).

Docker (no date) 'What is a Container? App Containerization Docker'. Available at: https://www.docker.com/resources/what-container (Accessed: 13 May 2020).

Ethnologue (2018) 'What are the top 200 most spoken languages?', *Ethnologue*. Available at: https://www.ethnologue.com/guides/ethnologue200 (Accessed: 17 March 2020).

Ferrero, J. (2020) 'FerreroJeremy/ln2sql'. Available at: https://github.com/FerreroJeremy/ln2sql (Accessed: 8 May 2020).

Fitzpatrick, P. (2020) 'Paulfitz/mlsql'. Available at: https://github.com/paulfitz/mlsql (Accessed: 12 May 2020).

Goldsborough, P. (2015) 'How to have multirow cells in Python table? - Stack Overflow'. Available at: https://stackoverflow.com/questions/28815268/how-to-have-multirow-cells-in-python-table/28817301#28817301 (Accessed: 8 May 2020).

Hadida, J. and sheljohn (2018) 'Print with colors in most shells (Python, standalone)', *Print colours in shell (gist.github.com)*. Available at: https://gist.github.com/Sheljohn/68ca3be74139f66dbc6127784f638920#file-colours-py (Accessed: 11 May 2020).

Hardesty, L. (2017) 'Explained: Neural networks', *MIT News*. Available at: http://news.mit.edu/2017/explained-neural-networks-deep-learning-0414 (Accessed: 4 April 2020).

Hochreiter, S. and Schmidhuber, J. (1997) 'Long Short-Term Memory', *Neural Computation*, 9(8), pp. 1735–1780. doi: 10.1162/neco.1997.9.8.1735.

Hwang, W. *et al.* (2019) 'A Comprehensive Exploration on WikiSQL with Table-Aware Word Contextualization', *arXiv:1902.01069 [cs]*. Available at: http://arxiv.org/abs/1902.01069 (Accessed: 13 May 2020).

IBM Cloud Education (2019) 'Relational-databases'. Available at: https://www.ibm.com/cloud/learn/relational-databases (Accessed: 25 April 2020).

International Standards Organisation (1992) 'ISO/IEC 9075:1992, Database Language SQL'. Available at: http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt (Accessed: 15 March 2020).

'Jgraph/drawio' (2020). JGraph. Available at: https://github.com/jgraph/drawio (Accessed: 17 March 2020).

Krebs, B. (no date) 'SQLAlchemy ORM Tutorial for Python Developers', *Auth0 - Blog*. Available at: https://auth0.com/blog/sqlalchemy-orm-tutorial-for-python-developers/ (Accessed: 17 March 2020).

Manning, C. D., Raghavan, P. and Schütze, H. (2008) *Introduction to information retrieval*. USA: Cambridge University Press.

Mozilla Developer Network (no date) 'Definition: REST', *MDN Web Docs*. Available at: https://developer.mozilla.org/en-US/docs/Glossary/REST (Accessed: 13 May 2020).

naver (2020) 'Naver/sqlova'. NAVER. Available at: https://github.com/naver/sqlova (Accessed: 12 May 2020).

NLP (no date) 'The Stanford Natural Language Processing Group'. Available at: https://nlp.stanford.edu/software/tagger.shtml (Accessed: 11 May 2020).

O'Neil, E. J. (2008) 'Object/relational mapping 2008: Hibernate and the entity data model (edm)', in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. Vancouver, Canada: Association for Computing Machinery (SIGMOD '08), pp. 1351–1356. doi: 10.1145/1376616.1376773.

Patel, Z. (2019) 'Zayd62/final-year-project', *GitHub*. Available at: https://github.com/zayd62/final-year-project (Accessed: 24 March 2020).

Patel, Z. (2020a) 'Zayd62/ln2sql'. Available at: https://github.com/zayd62/ln2sql (Accessed: 11 May 2020).

Patel, Z. (2020b) 'Zayd62/sqlova-test'. Available at: https://github.com/zayd62/sqlova-test (Accessed: 13 May 2020).

'Plantuml/plantuml' (2020). plantuml. Available at: https://github.com/plantuml/plantuml (Accessed: 17 March 2020).

'Pytest-dev/pytest' (2020). pytest-dev. Available at: https://github.com/pytest-dev/pytest (Accessed: 12 May 2020).

Python Software Foundation (no date a) '12. Virtual Environments and Packages — Python 3.7.7 documentation'. Available at: https://docs.python.org/3.7/tutorial/venv.html (Accessed: 11 May 2020).

Python Software Foundation (no date b) 'Python 3.7 Documentation'. Available at: https://docs.python.org/3.7/ (Accessed: 17 March 2020).

Richardson, L. (no date) 'Beautiful Soup: We called him Tortoise because he taught us.' Available at: https://www.crummy.com/software/BeautifulSoup/ (Accessed: 9 December 2019).

rstudio (2020) 'Rstudio/bookdown'. RStudio. Available at: https://github.com/rstudio/bookdown (Accessed: 13 March 2020).

salesforce (2019) 'Salesforce/WikiSQL'. Salesforce. Available at: https://github.com/salesforce/WikiSQL (Accessed: 29 November 2019).

scrapy (2019) 'Scrapy/scrapy'. Scrapy project. Available at: https://github.com/scrapy/scrapy (Accessed: 28 November 2019).

Silver, D. (2016) 'Deep Reinforcement Learning', *Deepmind*. Available at: /blog/article/deep-reinforcement-learning (Accessed: 4 April 2020).

sklearn (no date) 'Sklearn.metrics.f1_score — scikit-learn 0.23.0 documentation'. Available at: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html (Accessed: 13 May 2020).

sqlalchemy (2019) 'Sqlalchemy/sqlalchemy'. SQLAlchemy. Available at: https://github.com/sqlalchemy/sqlalchemy (Accessed: 28 November 2019).

SQLite Team (no date) 'SQLite Home Page'. Available at: https://www.sqlite.org/index.html (Accessed: 9 December 2019).

Zhong, V., Xiong, C. and Socher, R. (2017) 'Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning', *arXiv:1709.00103 [cs]*. Available at: http://arxiv.org/abs/1709.00103 (Accessed: 29 November 2019).

Zotero (no date) 'Zotero Style Repository'. Available at: https://www.zotero.org/styles?q=harvard (Accessed: 14 March 2020).