Seq2SQL: Translation of Complex Questions into SQL Queries

Shanelle L. Roman

Yale University

Abstract

The translation of natural language queries into structural SQL queries is an area of active NLP research. Most authors have built their models using the WikiSQL dataset released by Salesforce in 2017. However, the WikiSQL dataset contains simple SQL queries with very little format differentiation. As a result, this paper releases a new dataset with complex SQL queries that more accurately reflects the range of natural language questions used in everyday conversation. Using this new dataset, I then implemented the Seq2SQL model first proposed in the Zhong 2017 paper. Given the more complex SQL structure, I extended the model to predict additional SQL components.

Keywords: Machine Translation, Seq2SQL, WikiSQL

Expanding Seq2SQL to Complex Question Answering

To begin, this paper will cover the construction of the dataset, an outline of the methodology, results, and a discussion of further work to be done in this field. The overall goal of this project was to translate natural language questions into executable SQL queries. This is a nascent area of research within the field of Natural Language Processing, with many exciting business applications. Previous research has mostly been done using Zhong 2017's WikiSQL dataset. I mostly worked with Tao Yu and James Ma on constructing a new dataset that more accurately reflects real-world, complex questions and on creating a seq2SQL baseline model that reflects this more difficult problem. This research project drew heavily on the approach implemented in two previous papers. The first paper, "Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning" by Zhong et.al is the one that I based most of my final project on. The second paper is "SQLNet: Generating Structured Queries from Natural Language without Reinforcement Learning" by Xu and Song. Both of these papers, however, used the WikiSQL dataset, which is extremely simplistic and not real-world. As a result, my work was primarily extending the seq2SQL model to new data.

Dataset

As mentioned above, the dataset was based on the format and information contained in the WikiSQL dataset, but included more complex SQL queries. To begin, I will walk through the format of WikiSQL and then explain how and why the new dataset differs.

WikiSQL

The WikiSQL dataset is a large crowd-sourced dataset for developing natural language interfaces for relational databases that was created by the authors of the Zhong 2017 paper. It

consists of natural language questions that can be translated into simple SQL queries that contain the following basic components. The following is an example of a typical question used in the training data.


```
{
   "phase":1,
   "question":"who is the manufacturer for the order year 1998?",
   "sql":{
      "conds":[
         [
            0,
            0,
            "1998"
         ]
      ],
      "sel":1,
      "agg":0
   },
   "table_id":"1-10007452-3"
}
```


As you can see from above, the dataset does not actually include the text of the correct SQL query. Instead, it encodes the information using json fields for each of the SQL components. For the above question, the SQL query would be `"SELECT manufacturer WHERE order_year = 1998"`. The dataset splits each SQL query into three components: aggregation, selection, and condition. As a result, the dataset cannot be extended for more complex queries that include GROUP BY or SORT BY, for example. Each SQL query follows this format: `SELECT Agg? COL_NAME (WHERE COND1 (AND COND_NEXT)*)*`. As such, the SQL query structure was extremely limiting – there is only one aggregator and column selection, and the structure of the SQL query itself is predetermined. Moreover, the table_id field dictates which table the SQL query refers to,

which means that each SQL query only refers to one table. As a result, the algorithm only has to determine the column name from one existing table schema. However, as anybody who has ever used a SQL database knows, normally SQL databases contain smaller tables that are on discrete subjects. A typical SQL query, therefore, will use `JOIN` to combine information from multiple tables in a single database. As a result, a more real-world translation problem would introduce the question of picking the correct column from the right table, especially given that column names may have duplicates between tables. In summary, the WikiSQL construction of the problem can be defined as having a rigid problem definition, but better accuracies as a result.

New Dataset

In order to define a more real-world problem, we as a team constructed a new dataset. We found approximately 200 populated databases online, and we created questions with their corresponding SQL translations. Unlike WikiSQL, we include a much wider variety of components. For example, we introduce GROUP BY, HAVING, UNION, INTERSECTION, LIMIT, SORT, ASC and DESC. In addition, this new dataset also involves nested SQL queries inside the WHERE clause. The following is an example complex natural language question for our dataset: `“Select the name and price of all products with a price larger than or equal to $180, and sort first by price (in descending order), and then by name (in ascending order).”` Beginning with the SELECT component, the more complex dataset allows for multiple columns to be selected—in this case, name and price. Along with the multiple columns, the data can include multiple aggregations. For example, a query could include multiple aggregations for the same column (`min(col_1), max(col_1), avg(col_1)`), which also complicates the prediction problem. Additionally, we include `“*”` as an option, so all columns

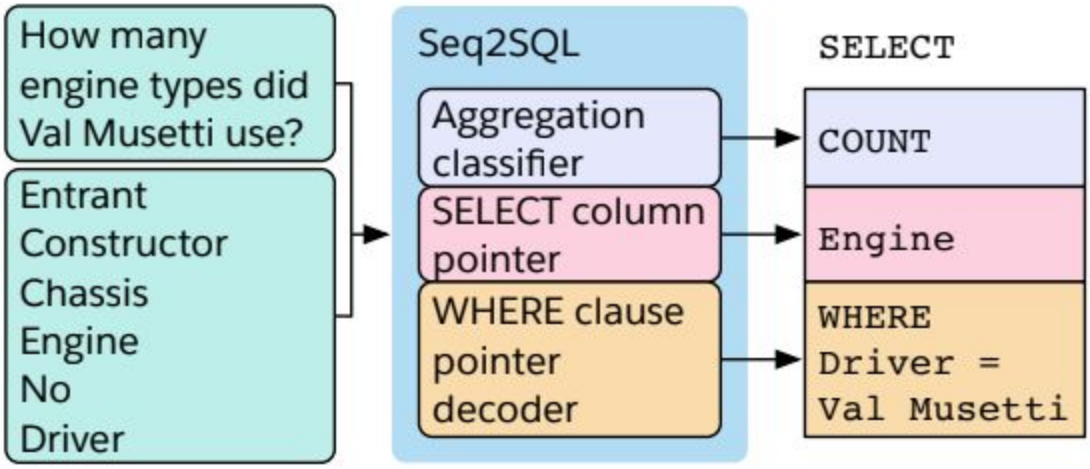
can be selected as well. Given the fact that the new dataset includes multiple tables per database and a single query can join multiple tables, in our encoding, we actually combined each of the tables and gave each column an unique index. As a result, it was almost as if the algorithm was selecting one column out a single table. When it comes to generating the JOIN component, each column index can then be mapped back to its table name and original index via the dataset. Additionally, when it comes to the operators for the condition of the WHERE clause, we included a wider variety. Instead of just "`>, <, =`," we included "`>, <, =, >=, <=, !=, not, between, in, like, is, exists.`" Additionally, whereas the previous SQL query only combined conditions in the WHERE clause with AND, we added OR as well. Moreover, we also included nested WHERE clauses, in which the condition was itself a complete SQL query.

Finally, moving onto the new components, we added many different ones to more accurately reflect the variety of questions asked in the real-world. For example, we included `SORT BY, ASC, DESC, LIMIT, GROUP BY, HAVING, INTERSECT, UNION.` Although some components mimic the format of the previous components, such as GROUP BY, others introduce a lot of additional complexity. For example, INTERSECT and UNION both combine two complete SQL queries. As a result, in order to accurately generate the SQL query, one would need to predict that UNION/INTERSECT is necessary and correctly splice the information into the two separate queries. Given this, the problem is substantially more difficult, which should frame the results below.

Method

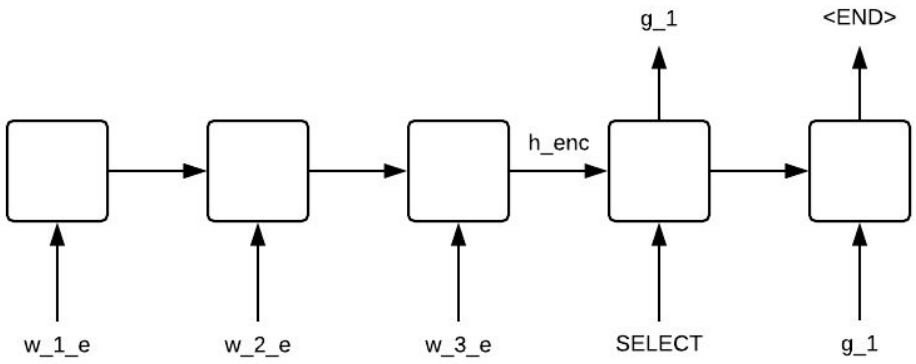
Original Seq2SQL

The original Zhong 2017 model is a little bit different given the simplicity of the previous task. As stated in the paper, the authors used an augmented pointer network to generate the SQL query, which generates a scalar attention score $\alpha\_\{s, t\}$ that represents the chance that SQL token s is the correct translation at each position t of the query. However, the Zhong 2017 paper split the generation of the SQL query into 3 separate neural network architectures. The following figure is a high-level explanation of the model from the original paper.



As one can see, the model takes as input the word embeddings of the natural language question and of the column names. The model takes those word embeddings and encodes them using a two-layer, bidirectional LSTM, which outputs the hidden state of the query represented by h^{enc}. Then, similar to the encoder-decoder network structure displayed in the figure below, each decoder component takes as input h_{s-1}, g_{s-1} and outputs g_s, where h_{s-1} is the previous hidden

state, g_{s-1} is the previous predicted SQL token, and g_s is the predicted token at timestep s. Then, it calculates $\alpha_{s,t} = W\ tanh(U \cdot g_s + V \cdot h_t)$, which can be interpreted as the similarity of the predicted output with the hidden encoded query at that time step. The predicted output is then $y_s = argmax(\alpha_s)$. Below is the graphic for the encoder-decoder sequence structure for the SELECT component.

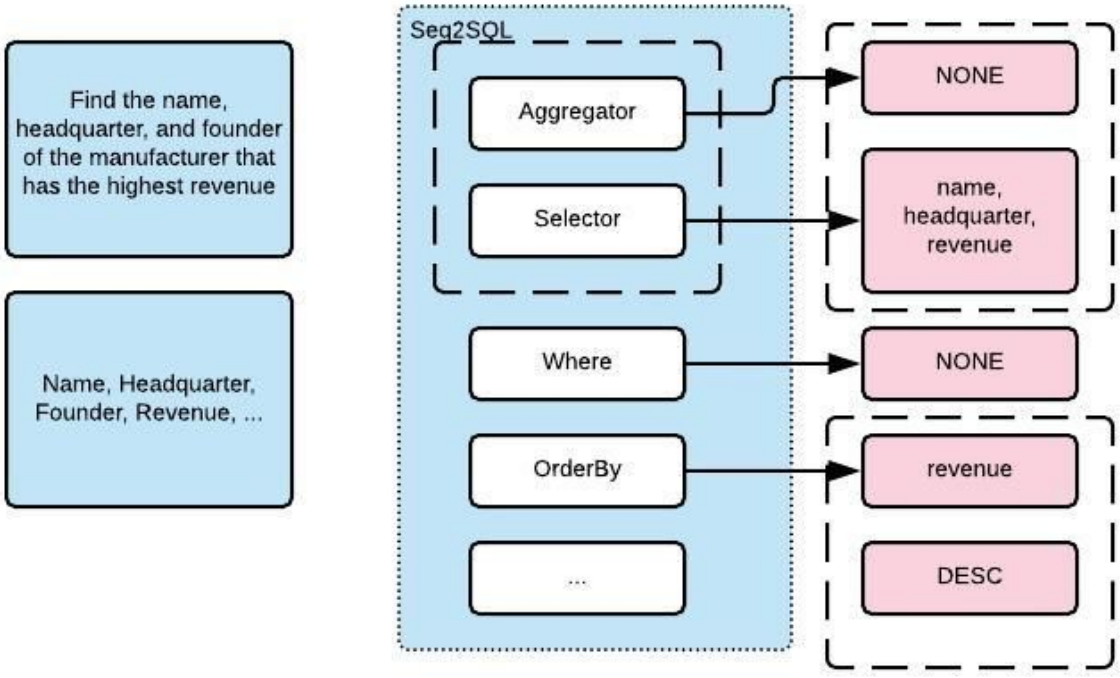


However, each of the different components has an unique neural network structure. The SELECT and AGG predictions were substantially simpler than the WHERE sequences because they are only predicting one column and one aggregator for each query. Given the number and complexity of the components in our task, we did not utilize the existing architecture. As a result, I will focus this section on explaining the WHERE condition component. The basic architecture is explained above, with some additional modifications. Because the algorithm is training a batch of 40 queries at a time, the input vector for the decoder has dimensions $[B,\ max\_len,\ max\_tok\_num]$, where B is the batch size, max_len is the maximum natural language question length, and max_tok_num is the maximum length of a SQL query. In

training, the input to each component of the decoder is the correct gold standard input, but in testing, the previous predicted output g_{s-1} is fed into the next component. In testing, the neural network stops predicting whenever the $< END >$ token is output. That is, in essence, the overall structure of the Seq2SQL model.

My Extension

Given the additional complications for our dataset, we restructured the Seq2SQL model by modifying the Condition Prediction module. Below is a graphic that details the overall structure of our model.



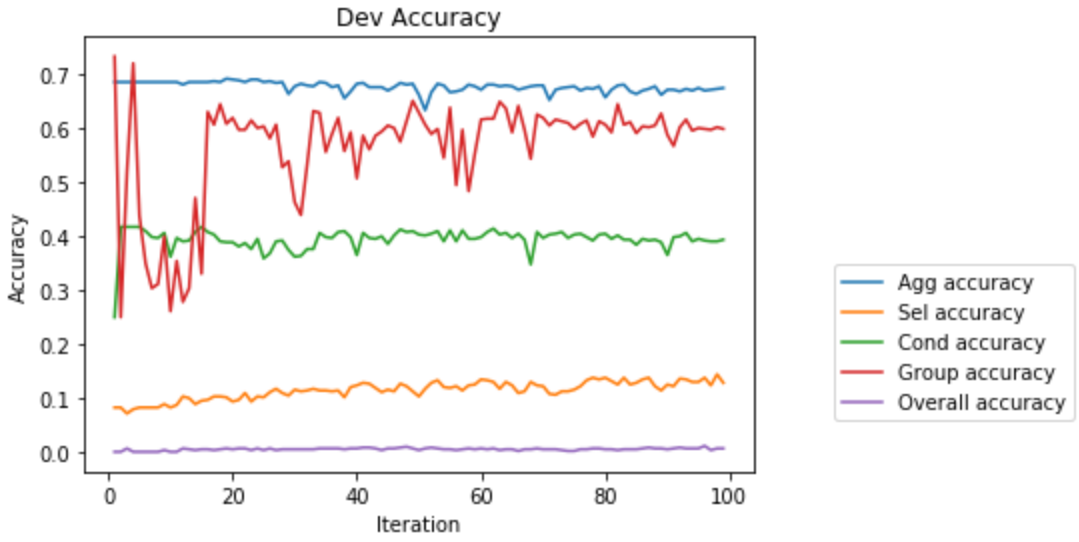
The model takes the same inputs as the original model. Whereas before only one column and aggregator was predicted, with any number of columns and aggregators possible for the new

dataset, it made more sense to output them as a sequence. In that vein, I also combined the prediction of the aggregators and selected columns into one model for generating the score in `model.forward()` because otherwise, it would be difficult to match a predicted aggregator to the selected column in question. In doing so, I created a superclass called Subsequence_Predictor that modifies the Condition_Predictor code. However, I only combined the AGG and SELECT predictions for the purposes of generating the scores. I later decoupled SELECT and AGG for the calculation of losses and generation of the queries. This eliminated some problems with the previous approach. First, it makes it easier to match up the AGG operator predicted with its corresponding SELECT prediction. Second, it is extensible for any number of columns and aggregator operators.

However, there were some differences from the Condition_Predictor code. For example, the choice of tokens that the algorithm could predict in the decoder section was different. For conditions, one could output "`WHERE, <UNK>, <END>, >, <, >=, ...`" as well as any of the column names or words in the original question. For predicting the columns selected, the outputs are necessarily different—"`SELECT, <END>,...`". As a result, I had to modify the indices that indicate the initial input and ending sequence—changes that I also applied for the prediction of the GROUP BY sequence.

Results

Overall, the results are not that promising, but that is to be expected given the additional complexity of the problem. Below is a graph of the validation accuracy for my model on the new dataset.



As you can see, the overall accuracy is, quite simply, abysmal—0.791139%. However, the individual accuracies show more promise on their own. The best individual validation accuracies are as follows—69.62% for aggregator, 14.24% for selection, 41.77% for conditions, and 73.42% for group by. The following is a table that compares the accuracies for my model against the original Seq2SQL for the WikiSQL dataset.

Accuracies	**Seq2SQL Original**	**Extension**
Total	49.5%	0.79%
AGG	90.07%	69.62%
SELECT	89.76%	14.24%
WHERE	60.62%	41.77%
GROUP BY	—	73.42%

Discussion

Given the low accuracies, there is a lot of room for improvement on the above model. For example, the highest accuracies were for AGG and GROUP BY, two components that appear relatively infrequently in the SQL queries. As such, it is worth investigating whether the high accuracies are due to the network predicting `[]` (nothing for those components). Additionally, the accuracies reported for conditions assume that we are interested in the value listed for the condition (e.g. col_val in `WHERE col_name where_op col_val`). However, upon further discussion with my colleagues Tao Yu and James Ma, we ultimately decided that we would leave the condition value out of the accuracy calculation for the sake of assuring fair results with the SQLNet calculation, which does not take the question tokens as input. Finally, the graph was produced after training for 100 epochs, so it might be worth it to train for a further 200 epochs to see if the overall validation accuracy improves.

Beyond the specific improvements listed, there are some larger structural changes that can be further explored. Additionally, the SELECT column accuracy was a lot lower compared to the original Seq2SQL model. In addition to choosing from more columns, there is also the issue of duplicate column names because the embeddings are numerical encodings of the column names in vector space. For the network, there is no difference between the `id` field in the `art` table and the `id` field in the `student` table. Currently, the network chooses randomly from the matching column names. However, by appending the table name to the column name and averaging the word embeddings, I hope that will improve the prediction accuracy. Finally, the listed accuracies are for the model without reinforcement learning applied. The last step will be to apply reinforcement learning to all of the listed components to improve all the accuracies.

References

Zhong, V., Xiong, C., Socher, R. (2017). Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *CoRR*, Vol abs/1709.00103

Xu, X., Liu, C., Song, D. (2017). SQLNet: Generating Structured Queries from Natural Language Without Reinforcement Learning.

Appendix

The code for this project can be found at the following github link:

https://github.com/shanelleroman/seq2sql.

The dataset can be found here: https://github.com/yangkai2g7k/nl2sql.

Given that this project is part of a paper that is going to be submitted for a conference, the github repositories will be updated past the submission of this project.

I also wanted to thank James Ma, Professor Radev, and Tao Yu for helping me at every stage during this project. I could not have completed it without you all!