Generating SQL Queries from Natural Language

Vatsal Patel and **Vartika Jain** and **Kshitij Patel** and **Sarth Kanani** and **Pragya Goel**
Department of Computer Science
University of Southern California, Los Angeles, CA 90007

Abstract

Relational databases store a significant amount of the world's data. However, accessing this data currently requires users to understand a query language like SQL. Thus, it is imperative to have intelligent interfaces for interaction with the data. This should be done to enable even non-expert users of SQL to make efficient use of the data. Generating executable queries from the question posed by the user has been a long-standing problem, which has been gaining much momentum lately. Traditional methods involve using sequence-to-sequence models but the generalized nature of such models does not exploit the full structure of a SQL query.
In this project, we explored the existing state-of-the-art model for generating SQL queries from natural language questions proposed by (Zhong et al., 2017) and observe the SELECT column, WHERE clause and Aggregate function parts of the SQL query. However, we implemented the new state-of-the-art models proposed by (Ikshu and Archit, 2018) for Aggregator function since it directly analyses the given Natural language sentences and (Xu et al., 2017) for SELECT and WHERE clauses since these are improved versions for both the operations.

1 Introduction

The objective of our project is to convert Natural Language sentences to SQL queries which form the backbone of Relational Databases Systems.
Relational databases provide a powerful way to store records and data, however, extracting these records requires skill and knowledge of query languages like SQL. Thus, to cater the query language processing without having a deep understanding of its semantics and syntactic structure, NLP plays an important role in enabling even non-experts to make efficient use of data and draw relevant inferences from the databases.
Novitiates are often overwhelmed by the concepts of structured queries, joins, and aggregate functions. NLP aims to ease those burdens on new and experienced users alike and to develop tools to generate SQL commands from natural language.
This project tries to explore and implement the techniques introduced by (Ikshu and Archit, 2018) and (Xu et al., 2017), solving the problem of retrieving information, so that natural language questions can be used to fetch the required data.
Once we are able to re-implement these ideas successfully, the non-experts and non-technical people will also be able to generate SQL queries from relational databases without having hands-on knowledge of the technology domain. This will help database users to draw inferences successfully from vast amounts of data and turn information into knowledge discovery successfully.

2 Related Work

The research on using neural networks for machine translation has been ongoing for around a decade and major improvements have been achieved in the area of parsing natural language to SQL recently by (Zhong et al., 2017), and (Xu et al., 2017). (Zhong et al., 2017) introduced seq2SQL, a deep neural network architecture combined with reward-based Reinforcement Learning for parsing natural language to SQL queries. It focused on dividing the query into three significant components: aggregator function, SELECT Column predictor, and WHERE clause decoder based on the natural language and a high-level database schema.
(Xu et al., 2017) further proposed SQLNet, a sketch-based architecture that makes use of the structure of a SQL query, and generates it from a dependency sketch which describes the dependency between different parts of the query. It uses a column attention mechanism for predicting the SELECT and WHERE clause columns which are improved versions of the original state-of-the-art models. After this, several approaches like (Yu et al., 2018), which approached the problem as a slot-filling task have been proposed to tackle the

problem.
The recent improvements have been proposed by (Ikshu and Archit, 2018), which focuses on improving the Aggregate operation of the SQL query by using the natural language question instead of using predicted SELECT column as described in the paper (Zhong et al., 2017). Mainly because the question will contain enough information required for the prediction of the aggregate operation and secondly, this avoids forced mis-predictions due to mis-predicted 'SELECT column'.
Our main focus in this project has been to re-implement the techniques of the 2 papers (Ikshu and Archit, 2018) for implementing the aggregate function of the SQL using natural language question and (Xu et al., 2017) for the SELECT & WHERE components using an attention-based mechanism to preserve the long term dependencies and the context of natural language.

3 Problem Description

As stated above, our model will be based primarily on research presented by (Zhong et al., 2017) who introduced seq2SQL which focused on dividing the query into three significant components: aggregator function (COUNT, MAX, SUM, etc), SELECT Column Predictor, and WHERE Clause decoder based on the natural language and a high-level database schema. Our implementation methods are based primarily on the research of the two papers mentioned above. Once we can fetch all the 3 components of the SQL query from the Natural language question, we can successfully build the required SQL query to query data and extract patterns and inferences from it.

4 Implementation Methods

4.1 Data Collection

The dataset of choice for this project is WikiSQL, introduced by (Zhong et al., 2017). It's a corpus of 80654 hand-annotated instances of natural language questions, SQL queries, and SQL tables extracted from Wikipedia. Our dataset is divided majorly into train, dev and test datasets where each dataset contains 3 types of files:-
(a) train.json- contains the Natural Language Question for which the query is to be synthesized and its SQL Structured elements.
(b) train.tables.json- contains the schema of SQL tables and columns and actual data samples for each SQL table.

Figure 1: A row in train.json


```
{
   "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"
}
```


(c) train.db- Used to access data via SQL database like Oracle.
A line in train.json file looks like Figure 1- The label for each example is the SQL query to be synthesized, represented as an index of one of the aggregation operators (NONE, MAX, MIN, COUNT, SUM, AVG) and the column selected, which is the index of one of the table columns.

4.2 Data Preprocessing

For the Data Preprocessing, our underlying architecture is dependent on the glove embeddings since it provides vectors for English language words. Additionally, we performed the following data cleaning steps to make our data more useful for the data model.
(a) We removed the '?' from the natural language questions provided in the train.json file.
(b) We tokenized the natural language questions to get the individual words in the questions.
(c) We converted the English words into lowercase since glove embeddings support lowercase words.

One important annotation we concluded is that we cannot remove **Stop Words** from the data since words like 'in','from' etc provide important information about the SQL query components. For instance, if we want to convert a question like 'What player played guard for Toronto in 1996-97?', then we need to have the 'in' stop word since it acts as a selector for the 'WHERE' clause of the query. Therefore, removing stop words will lead to losing components of the SQL query selector.
Another annotation we made is we cannot perform **Word Lemmatization** since it will lead to losing important information. For instance- if we have a Question like- 'List the names of people who

perform running, walking and gymming', then we cannot reduce running to run since this query will not list people who practice running.

Figure 2: Aggregator function model architecture

4.3 Model Design and Architecture

4.3.1 Aggregator Operator

For predicting the Aggregator operator of our SQL query, we have used the implementation as outlined in (Ikshu and Archit, 2018) by directly predicting it from the given question, in contrast to finding it using 'SELECT column' as described by (Zhong et al., 2017).

As we can see from Figure 2, we have focused on implementing a Bi-directional LSTM model for the Aggregator operator. We provide a word-to-index mapping for the given question E_Q as input to the embedding layer and calculate prediction probabilities over all aggregate operators as:

$$P_{agg} = \textbf{softmax}(W_2^{agg}\textbf{tanh}(W_1^{agg}E_Q + b_1^{agg}) + b_2^{agg})$$

where E_Q is of dimension d. Dimensions of W_1 is d x d, b_1 and b_2 is d each and that of W_2 is 6 x d. Finally, we take the **max** of **softmax** normalized predictions P_{agg} over 6 operators to get the predicted index of the operator and output one of the aggregator functions- {NONE, MAX, MIN, COUNT, SUM, AVG}. We have added an extra aggregator for padding in the list of operators.

It can be intuitively seen if we get a question like- 'How many goals Messy scored?' The correct operator for this question is 'SUM' since the question asks for 'How many'.

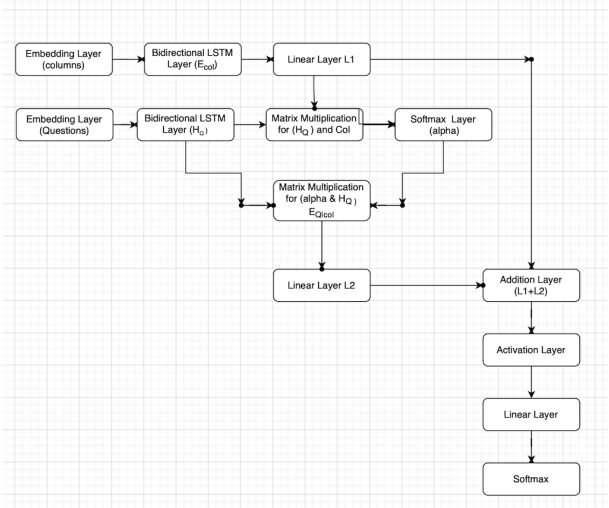
4.3.2 SELECT Operator

For predicting the SELECT operator, we have implemented the technique as outlined in (Xu et al., 2017). As shown in Figure 3, we are using a Bidirectional-LSTM model designed on attention-based mechanism which gives us the probability of a column occurring in the SELECT query given a question as input. Thus, we compute $E_{Q|col}$ instead of E_Q which is defined as:

$$E_{Q|col} = H_Q w$$

Here, H_Q is a matrix of d×L, where L is the length of the question and d is the dimension of the hidden

Figure 3: SELECT operator model architecture



states of the LSTM and w is the attention weights for each token in the question. w is calculated as :-

$$w = \textbf{softmax}((E_{col})^T W H_Q^i)$$

where H_Q^i indicates the i-th column of H_Q, and W is a trainable matrix of size d × d. Once we compute $E_{Q|col}$ value, we get the column-attention model as:

$$sel_i = (u_a^{sel})^T \textbf{tanh}(U_c^{sel}E_{col_i} + U_q^{sel}E_{Q|col_i})$$

$\forall i \in [1, ..., C]$ where C is number of columns. Here U_c^{sel} and U_q^{sel} are trainable matrices of size d × d, and u_a^{sel} is a d-dimensional trainable vector. The model will predict the column col_i as the SELECT column for the given natural language question that maximizes $P_{selcol}(i|Q)$ as-

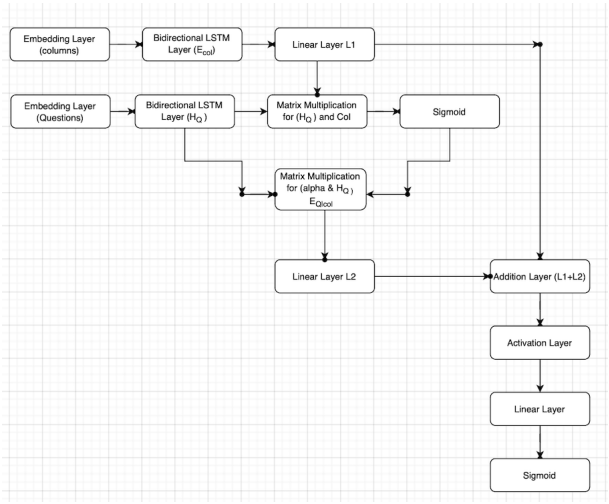
$$P_{selcol}(i|Q) = \textbf{softmax}(sel_i)$$

Thus, for example, consider the question "How many countries got 800 points?". It has token **points** which is relevant to predicting column 'Points' from a list of columns in a table (["Rank", "Member Association", "Points", "Group stage"]). So our model will give the highest value for column **Points** after applying **softmax** function and thus the found column for **SELECT** operator.

4.3.3 WHERE Clause

Similar to SELECT operator, we are using attention based-mechanism for finding the WHERE clause of our SQL query as described in architecture 4. We have implemented the model by referring same paper (Xu et al., 2017) as we did for SELECT operator. We are passing the column name and question

Figure 4: WHERE clause model architecture



embeddings to a Bi-directional LSTM model and computing the probability $P_{wherecol}(col|Q)$ as:-

$$P_{wherecol}(col|Q) = \sigma((u_a^{col})^T \textbf{tanh}(U_c^c ol E_{col} + U_q^c ol E_{Q|col}))$$

where σ is the sigmoid function, E_{col} is embeddings of column name, $U_c^c ol$ and $U_q^c ol$ are trainable matrices of size d × d, and u_a^{col} is a d-dimensional trainable vector. Thus, the decision to include a particular column in the WHERE clause can be made independently to other columns by examining $P_{wherecol}(col|Q)$ and finding column with maximum probability.

The above approach we've used for finding the **column** in WHERE clause. To get the **operator** in WHERE clause, we have 4 possibilities- (<, =, >, op). We're finding the operator value by the same mechanism which we've implemented for aggregator function. Here, 'op' is used as a None operator in the WHERE clause.

5 Experimental Results

5.1 Experimental Setup

In this work, we have focused on the WikiSQL dataset (Zhong et al., 2017). It's a corpus of natural language questions, SQL queries, and SQL tables as outlined earlier in detail under Data Collection section.

We are interested in the break-down results on different sub-tasks: (1) the Aggregator operator; (2) the SELECT column; (3) the WHERE column; and (4) the WHERE operator We implemented SQLNet using TensorFlow. For the baseline approach, we compared our results with the numbers reported by (Xu et al., 2017).

5.2 Results

Table 1: Accuracy scores for dev data

	dev data			
	Acc_{agg}	Acc_{sel}	$Acc_{wherecol}$	$Acc_{whereop}$
Seq2SQL(Zhong)	60.8%	60.8%	60.8%	-
SQLNet(paper)	90.1%	91.1%	72.1%	-
(ours)	88.9%	86.16%	73.3%	96.7%

Table 2: Accuracy scores for test data

	test data			
	Acc_{agg}	Acc_{sel}	$Acc_{wherecol}$	$Acc_{whereop}$
Seq2SQL(Zhong)	59.4%	59.4%	59.4%	-
SQLNet(paper)	90.3%	90.4%	70.0%	-
(ours)	89.5%	85.62%	72.6%	96.67%

Tables 1 and 2 summarise the overall results for query synthesis accuracy for the 4 components of the SQL query- Aggregate, SELECT, WHERE column and WHERE operator. We study the results of our re-implementation of SQLNet and compare it with the (Xu et al., 2017) accuracy and Seq2SQL baseline model by (Zhong et al., 2017). We observe that our implementation is giving better results than the Seq2SQL baseline model and similar results to SQLNet paper on both dev as well as test datasets. For the WHERE operator, these papers have not mentioned any accuracy results separately.

6 Conclusions and Future Work

In this paper, we worked on generating SQL queries from natural language questions. We exploited the inherent structure of SQL query to break it into parts that can be predicted independently. We re-implemented the techniques for SELECT and WHERE clauses by referring to paper (Xu et al., 2017) and (Ikshu and Archit, 2018) for Aggregate operator. Our model architecture is a composition of Bi-directional LSTM (used to encode question and column tokens embeddings), followed by multi-layered perception.

As part of Future Scope, we wish to expand our current research and apply it to include other complex SQL functions like join, GROUBY etc. We also plan to extend it and include multiple columns in the SELECT and WHERE operators as the scope of this research is limited to include only 1 column in the SELECT and WHERE part of the SQL query.

7 Division of Labour

- Data Collection and Preprocessing: Pragya, Vartika
- Aggregate operator: Kshitij
- SELECT operator: Vatsal
- WHERE operator: Sarth

References

Bhalla Ikshu and Gupta Archit. 2018. Generating sql queries from natural language.

Xiaojun Xu, Chang Liu, and Dawn Song. 2017. Sqlnet: Generating structured queries from natural language without reinforcement learning.

Tao Yu, Zifan Li, Zilin Zhang, Rui Zhang, and Dragomir Radev. 2018. Typesql: Knowledge-based type-aware neural text-to-sql generation.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning.