Auto-complete Natural Language Queries and Convert to SQL

Sneha Kedia, Supraja Krovvidi, Priyanshi Agarwal, Laxmi Garde, Priya Bannur
University of Southern California, Los Angeles
snehaked@usc.edu, krovvidi@usc.edu, pa92514@usc.edu,
lgarde@usc.edu, bannur@usc.edu

1. Introduction

A vast amount of information is stored in relational databases. However, accessing this information requires extensive knowledge of query languages used to communicate with the database. A lot of people are inexperienced in database querying, not everyone who wants to retrieve data from a database knows how to program SQL queries. We solved this problem by developing a deep learning-based model that converts Natural Language Queries (NLQs) to Structured Query Language (SQL) queries.

Within a search engine, query auto-completion can improve efficiency by predicting what the user might want to know, presenting options to the users based on a past or current query, pre-entering the query into a query box as the user types it, and suggesting the next word. The aim is to reduce query entry time and potentially prepare the search results in advance of query submission. Implementing auto-completion on NLQs can give users the freedom to ask questions about the database in natural language.

The proposed project aims to build an auto-completion feature for natural language queries and a tool to convert these NLQs to SQL queries using NLP. Such a system that converts the information retrieval questions to runnable queries can come in handy, especially in the banking and healthcare sectors. We aim to develop a novel architecture for NLQ to SQL conversion that has the flexibility for implementation on relational databases and can be fine-tuned on a specific domain for good results. Most of the existing model works on a database with one table. In this implementation, we propose a model for generating SQL queries from a natural language where we use the ACL anthology database, consisting of 13 tables. This model has better accuracy than the current state of the art model on a multi-table. Our solution is a combination of multiple Natural Language Processing methods from traditional heuristic-based approaches up to the most recent breakthrough, namely BERT.

2. Related Work

The task of converting a natural language query into SQL query is a mapping from plain simple text to a more structured and formal language which is used worldwide for database related operations. The base paper for our work: SyntaxSQLNet [1] uses the Syntax tree model for complex and cross domain NLQ to SQL generation tasks. The researchers of this paper have used the WikiSQL dataset [4] to train the Syntax tree and the Spider dataset [2] for evaluation. They encoded the table schema along with the encoded NLQ as input to the SQL query generator. Whereas, we have encoded the schema using natural number mappings and embedded it as a part of the model. The reason we did this is that our dataset is small and is limited to RDBMS. So, our schema remains the same throughout.

3. Method

3.1. Autocomplete Model

This model receives the input from the user and automatically completes the natural language sentences the user enters. The model returns the top five sentences closest to the user input. The sentences returned by the auto-complete model make use of TF-IDF approach to quantify

words taken as the input and compute a score for each word to signify its importance in the corpus. The TF-IDF values of every word in the query are used to calculate the cosine similarity, which helps in ranking the matching words in corpus and returns relevant matches.

3.2. Recommendation Model

The processed NLQs returned by the autocomplete model have placeholders. The idea behind using queries with placeholders to train this model is that we want to prevent it from returning the same query with different entity values. We have multiple approaches to pick entity values to replace these placeholders. We can add a layer of user interaction and take these values as input from the user or we can use a reinforcement learning model to pick these values based on the user's history. Here, due to lack of adequate resources like GPU and time, we have implemented a random value generator.

3.3. NLQ to SQL Converter

This model accepts the final NLQs with entity values and handles the problem of NLQ to SQL conversion by dividing it into three major sub-tasks, namely, the Table-column pair prediction, link prediction, and the condition prediction. In the first subtask, a decision tree is used that considers a combination of POS tagging and bigram model-based prediction at first, and if this fails, it makes use of fine-tuned BERT embeddings based predictors for the Table-column pair. A similar approach has been used for the third sub-task 'condition prediction', wherein a relation needs to be established between the specific columns and the named entities. To get the named entity, we used a combination of NER methods and POS tagging for our database, whereas a BERT based model is used for the condition prediction. The link is created by running BFS over the graph of table linkages or the database schema which returns the shortest path in SQL format with almost 100% accuracy.

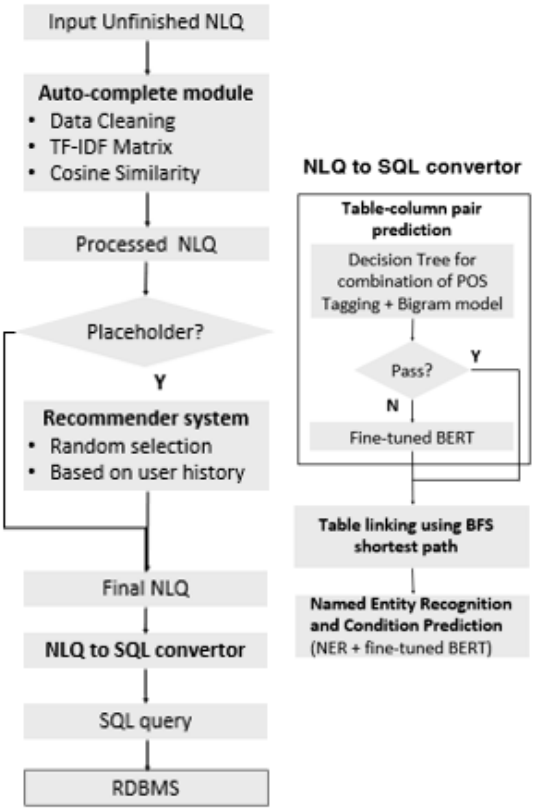

Figure 1: Flowchart for the proposed project

4. Experiment

4.1. Dataset Setup

The existing approaches tackling the problem generally involve training over a large dataset ([1], [2], [8]). We have trained our model on a dataset of multi-table relational dataset with a much smaller size as compared to the previous ones. We created a domain-specific dataset because it leads to a better performance than a model trained on a general domain dataset. We used the ACL-Anthology dataset [3]. It consists of 13 tables and 6 foreign key columns. On average a table contains 2 or 3 columns. AuthID, FieldID, PaperID, KeywordID, ConfID and AffiliationID are the foreign keys. Each of the tables contain around 5,000 rows.

4.1.1. Creation

We started by generating around 310 unique natural language queries (non-complex in nature, without HAVING, GROUP BY, ORDER BY operators) with placeholders. Each query is upto 25 words long and depending on

up to 5 tables. This dataset is used for training the autocomplete model.

4.1.2. Annotation

While creating each of these queries, we manually annotated them with the number of tables each query would need to reference.

Number of Tables	Number of Queries
One Table	145
Two Tables	23
Three Tables	118
Four Tables	18
Five tables	6

Table 1: Number of tables referenced

4.1.3. Paraphrase

We paraphrase each of the 310 NLQs to generate 310 unique pairs of NLQ-SQL queries.

4.1.4. Augmentation

As mentioned above, we have used placeholders instead of an entity. Entity here means a valid entry of each column. Placeholders are column names enclosed within $$. One such placeholder is $PaperID$ and its representative replacement would be any entity of the PaperID column. We randomly chose some 100 entities to replace each placeholder and augment it. This returned a total of 3100 NLQ-SQL queries pairs.

4.1.5. Tools

We used an open-source python library “querycsv” to open our csv data on a terminal interface. This library allows the annotators to see the schema and content of each table, execute SQL queries, and check the returned results. This library was extremely helpful for the annotation of complex SQL queries and is also used in the evaluation of the generated SQL queries.

4.2. Baseline Methods

4.2.1. NER

The natural language sentences given by the users contain entities such as names of places, authors, years, etc. We use the POS tagging approach to identify these entities. The input natural language sentence was tokenized, with the tokens being tagged with the type of POS. The proper nouns were identified as entity values; we also considered the some cases wherein they appeared within quotes or double quotes. We compared it with the database in order to obtain the column values. The output of this subtask was the named entity and column name pair which will further be used in the where condition space.

4.2.2. Table-Column Pairings

Once the NER sub-task returns an NLQ without the named entity, this sentence is then processed for identifying the column name that appears between the SELECT and FROM parts of the SQL query. For the column name identification, we first used Random Forest and XGBoost models with BERT-word Embeddings but they gave very low accuracy.

Model	Accuracy
Random Forest	57 %
XGBoost	65 %
POS Tagging with close match	80 %
POS Tagging with Lev. Distance	98 %

Table 2: Models used for table-column pairings

We found that the column names usually exist in some or the other word forms of the column names and are usually tagged as either nouns, proper nouns or verbs. Hence, we identified the column names from the sentences by tokenizing the natural language sentence and processing the POS tags corresponding to the tokens. We then matched the terms with the column names using the Levenshtein distance ratio and those with a threshold value of 0.55 turned out to hold true for most of the iterations.

4.2.3. Link Prediction

Many of the questions deal with data involving two or more tables. The SQL query requires these linkages between the table names in order to process the query for the given information. If the mentioned link is incorrect, the query falls short of information to look for the required condition. Hence, the link prediction task is key to the SQL query structuring task, and needs to be as accurate as possible. The link is prepared based on the database schema, which is represented as a graph with nodes as the table names and the edges denoting whether a particular table node has some column common with another table node. This link prediction task requires to have inner joins on the table names in the SQL query.

4.2.4. Condition Prediction

This subtask takes a deep learning approach in order to find a correspondence function between the natural language sentences and the conditions, i.e., the operator to be used in the 'WHERE' such as <, >, >=, <=, ==, etc. We fine-tuned BERT-Embeddings on the dataset and predicted the condition from the sentence quite accurately. A similar classifier was fine-tuned for predicting whether the query would contain 'COUNT' or the column name in between 'SELECT' and 'FROM' terms.

4.3. Evaluation Protocols

1. Exact Match: The percentage of the test set in which the predicted SQL query is the same as the actual SQL query.
2. Semantic Match: The percentage of the test set in which the predicted SQL query is semantically the same as the actual SQL query.
3. Query Distance: The exact match and the semantic match accuracy reveal only the percentage of exacts and not how close our prediction is. Whereas, query distance allows us to comment on the closeness of match based on the number of edges between the given columns in the BFS tree of the Schema graph.
4. Zero-result Rate: It determines the rate of queries returning no results at all. As far as the scope of this project is considered, zero-result rate plays a key role as our dataset is small and cannot provide results for queries outside the used domain.

5. Results and Discussion

The dataset is partitioned into training and testing sets. The query predictor scored 86.67% exact matches and 96.67% on semantic matches. The query distance measured upto 1.132 which is about 93.40%. Overall we observed a satisfactory performance on the dataset and believe that the project has promising scope of expansion.

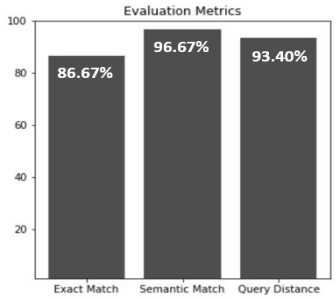

Figure 2: Evaluation Metrics for our model

6. Conclusion

As compared to the existing architectures, our approach is customized for a specific database and hence giving good accuracy. The baseline NLQ-SQL model based on SyntaxSQLNet, which is a syntax tree model for cross-domain tasks, in comparison to that we encoded the relational schema (natural numbers mapping) and embedded it as a part of the model. The model implemented by us offers flexibility for implementation on databases with varied schemas and can be fine-tuned for accurate results. For future work, we plan to create models that predict complex queries with clauses like HAVING, GROUP BY, ORDER BY and nested queries beyond 5 table linkages.

7. Team Responsibilities

1. Dataset annotation: all team members.
2. Auto-complete module implementation: Priya Bannur, Priyanshi Agrawal, Laxmi Garde.
3. NLQ to SQL module implementation: Sneha Kedia, Supraja Krovvidi.
4. Integration and testing: all team members
5. Poster work: all team members
6. Report work: all team members

8. References

1. Yu, T., Yasunaga, M., Yang, K., Zhang, R., Wang, D., Li, Z., and Radev, D. (2018a). SyntaxSQLNet: Syntax Tree Networks for Complex and Cross-DomainText-to-SQL Task. arXiv preprint arXiv:1810.05237
2. Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., Ma, J., Li, I., Yao, Q., Roman, S., Zhang, Z., and Radev, D. R. (2018b). Spider: A large-scale human- labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In EMNLP.
3. Singh, M., Dogga, P., Patro, S., Barnwal, D., Dutt, R., Haldar, R., Goyal, P., and Mukherjee, A. (2018b). CL scholar: The ACL anthology knowledge graph miner. CoRR, abs/1804.05514
4. Zhong, V., Xiong, C., and Socher, R. (2017). Seq2sql: Generating structured queries from natural language us- ing reinforcement learning. CoRR, abs/1709.00103
5. Tahery, S., & Farzi, S. (2020). Customized query auto-completion and suggestion—A review. Information Systems, 87, 101415.
6. Xiao, C., Qin, J., Wang, W., Ishikawa, Y., Tsuda, K., & Sadakane, K. (2013). Efficient error-tolerant query autocompletion. Proceedings of the VLDB Endowment, 6(6), 373-384.
7. TF-IDF from scratch in python on a real-world dataset (2019): https://towardsdatascience.com/tf-idf-for-document-ranking-from-scratch-in-python-on-real-world-dataset-796d339a4089
8. Wang, P., Shi, T., and Reddy, C. K. (2019). A translate-edit model for natural language question to sql query generation on multi-relational healthcare data. arXiv preprint arXiv:1908.01839