NILE: An Interactive Natural Language Interface for Relational Databases

Anil Shanbhag, anils@mit.edu

December 15, 2015

Abstract

There is an increasing push towards making databases more accessible. A natural language interface for databases (NLIDB) is considered as the ultimate goal for a database query interface. Existing approaches are tailored to work in specific circumstances. In this report, I describe NILE, a prototype NLI for relational databases which makes no assumptions about the database schema. NILE uses an off-the-shelf parser to construct dependency parse tree, constructs questions to ask the user about domain-specific terminology, uses a classifier to take local actions on the parse tree to make it resemble SQL and in the final pass generates a SQL query from it. By these means, a logically complex English language sentence is translated into a SQL query, which may include joins and nesting. [1]

1 Introduction

SQL has been the standard language used to query databases. While expressive and turing-complete, SQL is difficult to write especially for non-technical users. There has been an increasing push to make databases more accessible by making more user-friendly query interfaces.

Natural language is the most natural way to express queries. There has been a lot of interest in creating NLI for databases. However they have not been adopted due to low precision, that is user gets wrong results. They are primarily rule-based systems. Luke and Zettlemoyer[2] use a PCFG-based approach to generate logical expressions for natural language queries given training data on the same database. However in the real world it is unlikely to get training data for the database being queried. Li and Jagadish[1] use transformations on the dependency parse trees to generate trees resembling SQL. They impose restrictions on the way user specifies the query, cannot handle domain specific terms and use a rule-based approach.

[1] Github repo: https://github.mit.edu/anils/nile

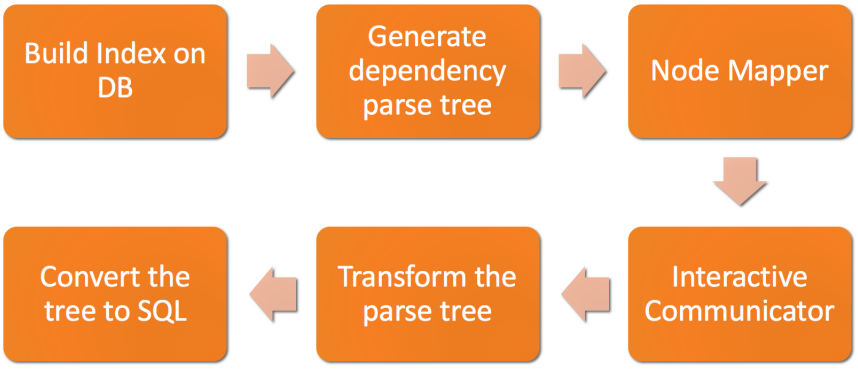

Figure 1: Overview of NILE

In this project, we explore the following problem setting. Given training data consisting of natural language query mapped to SQL on a database(s), generate SQL for a natural language query on a new schema. Our approach is to do transformations on the dependency parse tree (similar to [1]). We describe two approaches to do it, one is a rule-based approach and which takes actions based on logistic regression model. We use the user to disambiguate terms in the query.

2 System Overview

Figure 2 depicts the system architecture of NILE. The user selects a database and submits a natural language query as input.

The first step in the process is to build an index on the database elements (lexicon). The lexicon contains all the table names and column names. It also contains all the string values of the different columns in the database. The index is essentially a mapping from string to entity type.

We use the Stanford parser to generate the dependency parse tree of the user query. The tree hence created has nodes of the form (token, POS tag, arc relation to parent). The node mapper tries to map nodes to elements in the grammer using the index built earlier. The interactive communicator generates questions to ask to the user to resolve ambiguity / ask meaning of some terms in the query. The tree transformer modifies the structure of the parse tree to make it resemble the way SQL query would read. Finally the SQL generator uses the final parse tree to generate the SQL. The last 4 components are explained in greater detail in the subsequent sections.

3 Node Mapper

To understand the query from the database perspective, the nodes in the parse tree need to be mapped to SQL components. We map each node to one of the following 4 components:

- Select Node(SN): Represents the keyword SELECT.
- Name Node(NN): Relation / Column name
- Value Node(VN): Value in a column
- Function Node(FN): Function that transforms a relation
- Operator Node(ON): $=, \leq$,etc

The select node is mapped to one of set of keywords that occur first in the query (eg: what, which, return).

The first step in the process creates an index on database elements (lexicon). For each noun(NN*), verb(VB*), adjective(JJ*), we use the lexicon to see if the token maps to any value in the index. Tokens mapping to table or column names are labelled as name nodes and tokens mapping to column values are mapped to value node. Value nodes can be from a set of predefined words (like average, sum) or can be domain specific terms.

There might be tokens which are occuring in the plural form. For example, if query starts with "which cities ", cities here refers to city. We use NLTK library lemmatize to convert to singular form and then check in the index.

4 Interactive Communicator

NILE is meant to be an interactive system where user asks the query and system returns the results. Since the system does not have any prior training data on the queries related the db, ambiguity arises in the meaning of terms in the query.

The first form of ambiguity is token mapping to multiple values in the database. For example, in the geo database, the word 'Mississippi' can map to the state Mississippi or the river Mississippi. In such a situation, the node mapper would have stored the list of all possible options for the token. The interactive communicator asks the user to choose from the set of options and set the chosen one as the right options.

The second form of ambiguity is from domain specific terms. For example, in the geo database, there is no occurrence of the word 'density'. There is however for a given state, its area and its population. NILE extracts the context of the word from the parse tree. For example in the query 'what is

population density of texas ?', the context is 'population; name node' and 'texas; value node'. It then asks the user to express the meaning of density in terms of state.population and state (simplifies to just state). The user returns expression 'state.population / state.area'. The context is important as the same term may mean different things in different contexts (eg: city population density vs state population density). Simple table values get mapped to name node while expressions get mapped to function nodes.

5 Tree Transformer

The dependency parse tree is transformed to make it resemble SQL. We try to make it resemble the following grammar:

```
ROOT -> Select_Clause (Predicate)*
SelectClause -> SN + Expr
Expr -> (FN NN) | NN
RExpr -> Expr | VN
Predicate -> ON + (Expr RExpr) | ON + RExpr
```

Here '+' means parent child relationship, '*' means there could be multi children of the same form. Currently the grammar doesn't allow arbitrary nesting, this is a limitation of the current SQL generator.

We experimented with two approaches for doing the tree transformations. The first approach is a rule-based approach which are applied at each node in the tree. Some of the rules encoded are:

- Handling compound: Compound terms need to combined and simplified sometimes. For example: 'columbia river', columbia occurs as compound to river. 'border colorado' also occurs as a compound. This rule checks the types and combines the two into a single token if they are of the same type.
- Moving predicate out of select clause: Sometimes a predicate in the SQL query gets attached below select clause itself (eg: population of colorado, of colorado gets attached to population). We use the preposition (IN) tag to remove it out.
- Reordering select to root
- Order function nodes and operator nodes to conform to grammer. (eg: reordering population - density - texas to density - (population + texas)).
- Deleting nodes: Delete prepositions, determinants and verbs without any children.

The second approach I explored is using logistic regression to automatically do the transformations. There are three basic transformations that need to be done: swap with parent, combine with parent or make them siblings. These three transformations are sufficient to go from any parse tree to any other parse tree. The logistic regression is trained on training examples from a dataset seperate from the one used to test on. The set of features used are (POS Tag, arc, node mapping) for each of parent, leftmost sibling and rightmost sibling. Each of entries is a one hot vector.

6 SQL Generation

Once we have generated the final parse tree, we convert it to SQL by using the schema information from the database. First we construct a schema graph, where nodes are the tables and there is an edge between two tables if there if a foreign key relationship between them. For example, city contains 'stateid' which maps to state. So there is an edge between city and state table.

The select clause and each of the predicate blocks are individually generated. This a rule-based system, where for each predicate block it finds the relevant pattern and generate the where clause based on that. From the select clause, the target table is found. We traverse the schema graph (via BFS) to find the path from the predicate's return table to the target table. See Section 7 for a concrete example.

The SQL generation is not feature complete, it handles a limited subset of SQL. In particular it does not generate 'GROUP BY' clauses and hence doesn't support aggregation.

7 Full Example

Here we present the actual dialogue seen by user when he submits a sample query. A relatively simple query which shows the different aspects of the system is chosen. Consider user submitting the following query:

```
What are all the rivers in colorado ?
```

Initial Tree: Each node is (token, POS Tag, Arc Tag, Node Map)

```
what WP ROOT SN ( are VBP cop D , rivers NNS nsubj NN( all PDT det:predet UD ,
the DT det D , colorado NN nmod VN( in IN case UD  )  )  )
```

Question Asked:

```
What is colorado ? Options: [['colorado', Column(name=name, table_name=state)],
['colorado', Column(name=name, table_name=river)], ['colorado springs',
Column(name=name, table_name=city)], ['colorado river',
```

```
Column(name=lowest_point, table_name=highlow)] ? Ans: 0
What is all in terms of Table(name=river, columns=[...]) ? Ans: river
```

Transformed tree:

```
what WP ROOT SN ( rivers NNS nsubj NN, in IN case ON ( colorado NN nmod VN  ) )
```

SQL Query:

```
SELECT river.* FROM river WHERE river.id = (SELECT flows_through.river_id FROM
state INNER JOIN flows_through ON flows_through.state_id = state.id WHERE
state.name = colorado)]
```

8 Experiments

To test the system, we used two datasets 'Geo880' which contains 880 questions related to a geo domain and 'Jobs640' which contains 640 questions related to the jobs domain. Each of these came as set of prolog facts which needed to be converted to a database schema and values uploaded into a database. Also the queries come as mapping from natural language to prolog expressions. A good amount of time was spent in converting and verifying correctness of the queries generated. Finally all the facts where added into a Postgres database and dataset of queries mapped to SQL generated.

The system currently does not handle aggregations, hence we select a subset of 379 queries from the geo dataset. There were around 40 queries which did not fit the db schema. These were questions relating to USA or were wrong questions for example city area when this information doesn't exist in the database. The rest of the queries contained some form of aggregation. 30 queries were manually converted to final parse tree form, each node in the tree served as a training example for the LR appraoch. We achieve the following results.

The baseline chosen is work by Luke and Zettlemoyer[2]. Luke and Zettlemoyer use a PCFG-based appraoch assuming given training data on the same database. Note that the baseline is on the entire set of queries. One observation from the dataset is that most queries are very repetitive. For example there are 8 queries of the form 'what is the area of ¡state¿ ?'. This helps a PCFG based approach do well when more half the queries are used as training.

Method	Precision	Recall
Approach 1: Rule Based	98.22	60.4
Approach 2: LR	100	31.13
Zettlemoyer and Collins	96.25	79.29

Figure 2: Comparing the different approaches on geo dataset

Here recall is defined as the number of queries converted vs total number of queries. Precision is the fraction of the queries correct among the set of queries translated. The rule based approach has low recall due to the number of rules encoded into the system. For example, it does not correctly translate 'through which mississippi runs' as a predicate and doesn't make 'Austin, Texas' as one node.

The logistic regression approach is currently trained on a small set of examples. Since he feature vector is large and sparse, it is unable to perform well. The next step is to use the final parse trees generated using the first approach as training data for the second approach. This should significantly improve the recall of approach 2.

9 Discussion

Though no talked about in the above sections, the initial idea was to use reinforcement learning based approach to do the transformations. The parse tree being the environment, action being a transformation. Encoding the tree as a vector is non-intuitive and the number of states is very large. Hence did not proceed with this approach, thought I did spend time fiddling with RL-Glue which is a framework for doing RL.

Things to do as future work to improve the system are (not done due to lack of time):

- Using approach 1 to generate mapping of query to final parse tree, then train approach 2 on this.
- Supporting aggregation.
- Reducing the number of queries asked to the user based on automatic inference. Eg: 'return population of colorado', here colorado can't be a river.

10 Conclusion

In this project, we explored a way to convert natural language queries to SQL in the absence of training examples given on the db schema. It uses user interaction to disambiguated terms in the query and uses a rule-based/logistic regression approach to transform the dependency parse tree to make it resemble SQL.

References

[1] F. Li and H. Jagadish. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment*, 8(1):73–84, 2014.

[2] L. S. Zettlemoyer and M. Collins. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. *arXiv preprint arXiv:1207.1420*, 2012.