1 Introduction

After we have trained a deep neural model on a database with specific knowledge domain, the model has already gained some general rules to enquiry any database. With these shared knowledge, when we encounter a new unseen database with different domain, we do not have to re-train a model from scratch, but can transfer the model we already trained and apply on the new database with domain-specific fine-tuned training.

For example, if we have already trained a deep neural query interface on one employee information table, which contains the information of employees’ names, ages, genders, salaries, etc., and the query interface is well able to deal with query such as ‘How old is George’ or ‘what is the age of George’, then the query interface should be able to deal with some queries on a different table, such as ‘what is the population of Beijing’ on a geographic database, because the trained model has already seen knowledge of ‘what is the <predicate> of <subject>’, which could be shared to all database during enquiry. However, if we use the vanilla seq2seq translation model as the deep neural natural language interface (NLI), with raw query as encoder input and corresponding logical forms as output (decoder input during training), it tends to over-fit on training domain knowledge, but when comes to a new and unseen database, it would be very difficult to semantically parsing the related query to corresponding logical forms.

Therefore, we need to train a deep neural parser that could learn the template of the query, where some of the specific tokens indicating the fields and values are abstracted, and then map the query template to corresponding logical template which is machine compatible, such as SQL commands.

To do that, first we develop a tagging algorithm that based on dependency-tree parser, tagging each token in original query with a label like ‘<field:0>’,’<value:1>’ or ‘<nan>’, indicating that the token corresponds to a field name, a value, or neither, respectively. By doing so, we have gained latent information regarding the query beforehand, and we can pass both the query and the tag into the NLI as input. We have two ways to integrate both query and tag, named Parallel mode and X mode, respectively.

Since the SQL logical template generated by decoder also contains placeholder like ‘<field:0>’ and ’<value:1>’, we further develop a new seq2seq-based deep neural network called Seq2seqBridge, which could enable the information sharing and synchronic updating between the tags/placeholders in encoder and decoder, significantly improving the accuracy of decoder generating the SQL logical template. (Related Work: comment on pointer network and attention-based copying)

The datasets we have been using to train and evaluate our model are Wikitable, GeoQuery, and Overnight.

2 Related Work

(original written)

2016 seq2seq

2017 cross domain

[] use natural language as an intermediate representation to transfer knowledge across domains; seq2seq model for paraphrase. Domain adaptation protocol. We will use the following protocol: (1) train a paraphrase model using the data of the source domain, (2) use the learned parameters to initialize a model in the target domain, and (3) fine-tune the model using the training data of the target domain.

Herzig and Berant (2017) have explored another direction of semantic parsing with multiple domains, where they use all the domains to train a single semantic parser, and attach a domain-specific encoding to the training data of each domain to help the semantic parser differentiate between domains.

3 Pipeline: converting queries into SQL commands

(figure Pipeline)

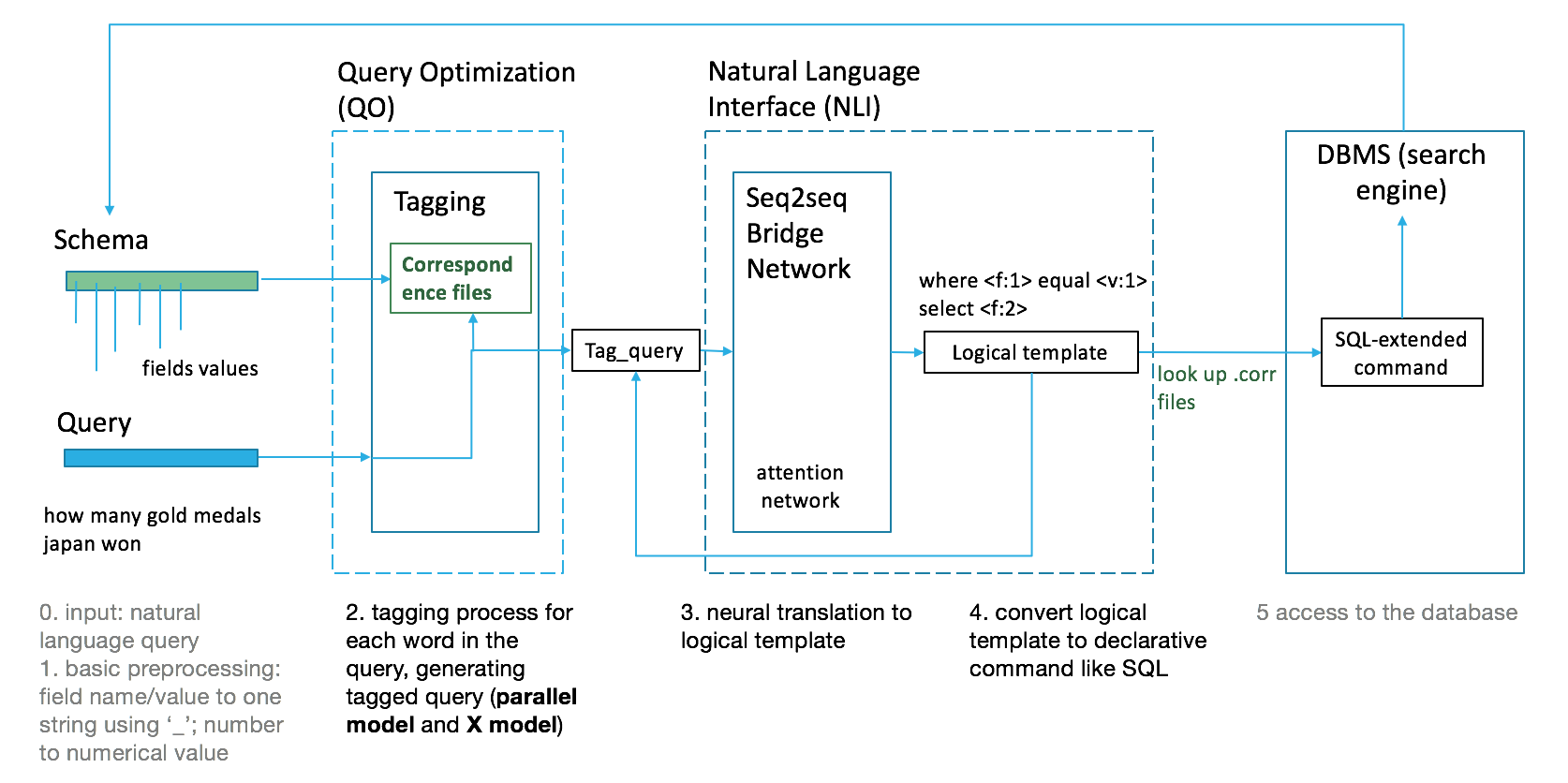
To build a natural language interface which is general and adaptable to all different database, we establish a pipeline that contains several stages:

Query optimization --- to identify the words in the query which indicate the information of the schema and tag them. We use designed features to represent tags, and record the correspondence information of fields and values which appears in the query. (tagging query with placeholder based on embedded schema information,) we will also store the correspondence information in memory during the inference stage, which could be easily retrieved when finally accessing the database.

Natural Language Interface to Database (NLIDB) --- We have also developed a new attentive sequence-to-sequence model Seq2seqBridge that is general and adaptive to knowledge bases. The Seq2seqBridge model is used to semantically parse the tagged query to the logical template based on SQL command. Parallel model and X model

Database enquiry --- during inference, the SQL logical template generate by the Seq2seqBridge model will be deterministically recovered to SQL command with the information of the correspondence information saved during the Query Optimization stage

Cross-database transfer learning --- schema (field names, values), several possible query sentences (specifically in this domain), fine-tuned training on augmented data based on the given information.

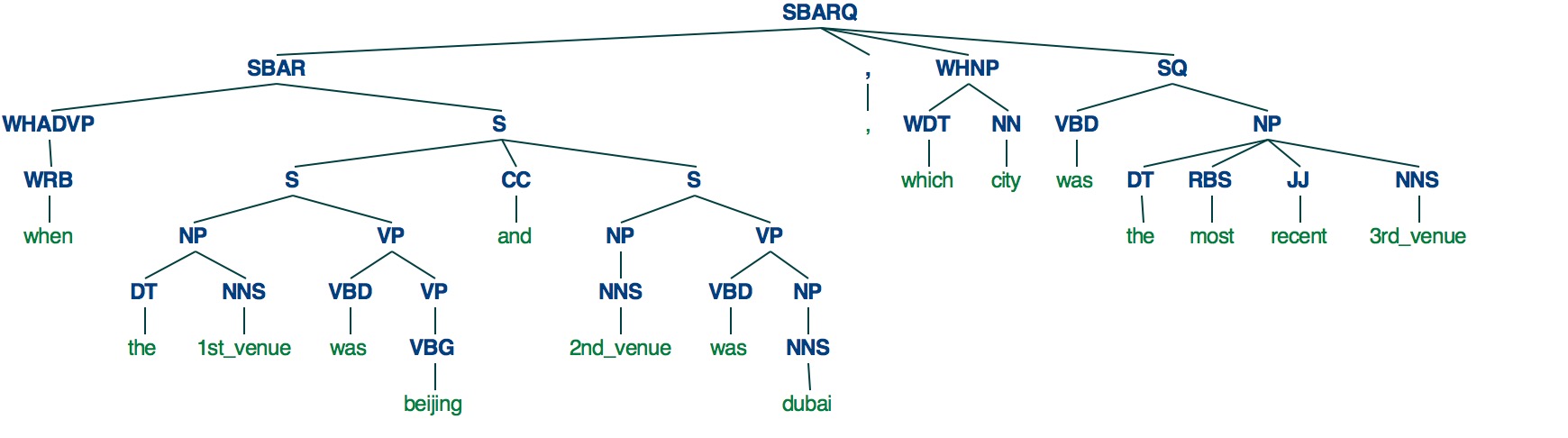


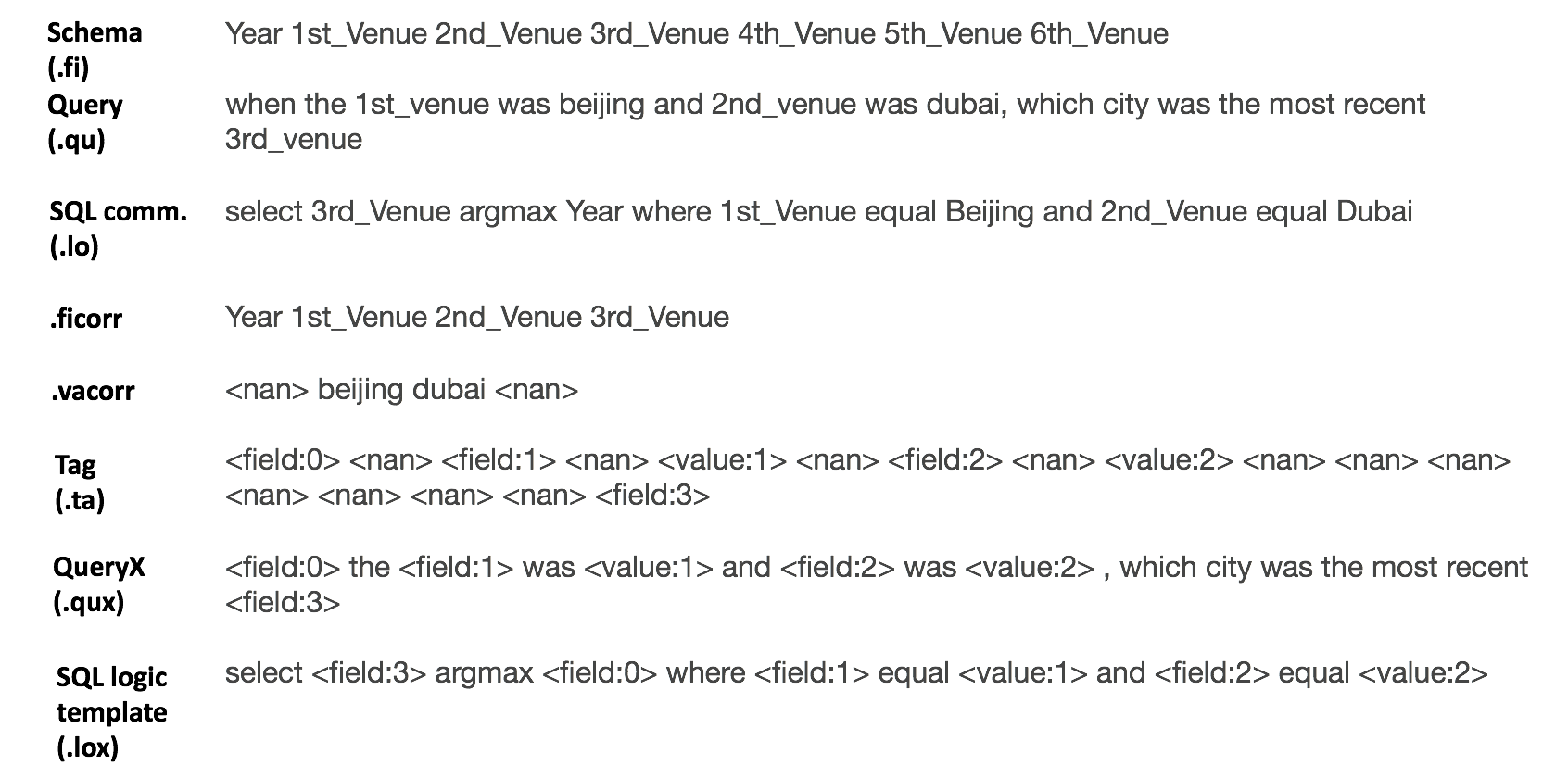
3.1 Query optimization: Dependency-tree-based Tagging

To generalize the NLI to different database and make it easy to adapt to other domains of knowledge base, we decide to firstly tag the tokens in the query with identifiers that indicate the knowledge of the querying schema. We want to identify each token in the query with either ‘<field>’, ‘<value>’, or ‘<nan>’, and for field with its corresponding value appeared in the query, we need to use the same id=0,1,2… to group them and distinguish from other fields. While we are tagging the query, the information of which fields in the schema appeared will be recorded sequentially, and so as the related field values appeared. Therefore, the final tag for a token could be like ‘<field:0>’, ‘<value:1>’, or ‘<nan>’. Later we also embed more information to the tag, such as the value type of the field, i.e. ‘num’, ‘string’, ‘ordinal’… and we have the extended version of tag as ‘<field:0;num>’, ‘<value:1;string>’, etc.

It is relatively straightforward to identify the tokens that correspond to field names or values; for example, we can use bloom filter by storing the value domain of each field (which serves as a prior knowledge) and name entity recognition for words not yet included into the value domain.

However, for fields with overlapping value domains, e.g. fields with numerical values, it’s difficult to attribute the value with the same ID to the correct field. With help of dependency parser, we group together the value and field with the lowest common ancestor in the dependency tree. For example, the query “when the 1st\_venue was beijing and 2nd\_venue was dubai, which city was the most recent 3rd\_venue” is complicated, as the fields appeared in this query ‘1st\_venue’ ‘2nd\_venue’ ‘3rd\_venue’ correspond to a huge value domain with different cities names, and we need to attribute ‘beijing’ and ‘dubai’ to the correct field names (in this case ‘1st\_venue’, ‘2nd\_venue’, respectively). Nearest neighbor method typically does not work quite well in natural language processing problem, due to the hierarchical and sequential structures of queries and sentences. To capture the relationship between a value and its correct field, it would be of great help to analyze the dependency tree of the query.

 We use the pretrained StanfordParser to generate the dependency tree of the query (figure), and for each value, we find its lowest common ancestor (LCA) with a possible field; the field with the deepest corresponding LCA is the one the value is attributed to. For example, from the figure we can see that the LCA of (‘beijing’, ‘1st\_venue’) is ‘S’ with a depth of 4 (assuming the depth of root is 1), the LCA of (‘beijing’, ‘2nd\_venue’) ‘S’ with a depth of 3, and the LCA of (‘beijing’, ‘3rd\_venue’) ‘SBARQ’ with a depth of 1; thus we attribute ‘beijing’ to ‘1st\_venue’ which paired with the deepest LCA. Therefore, the final tagging corresponding to the original query would be ‘<field:0> <nan> <field:1> <nan> <value:1> <nan> <field:2> <nan> <value:2> <nan> <nan> <nan> <nan> <nan> <nan> <nan> <field:3>’.

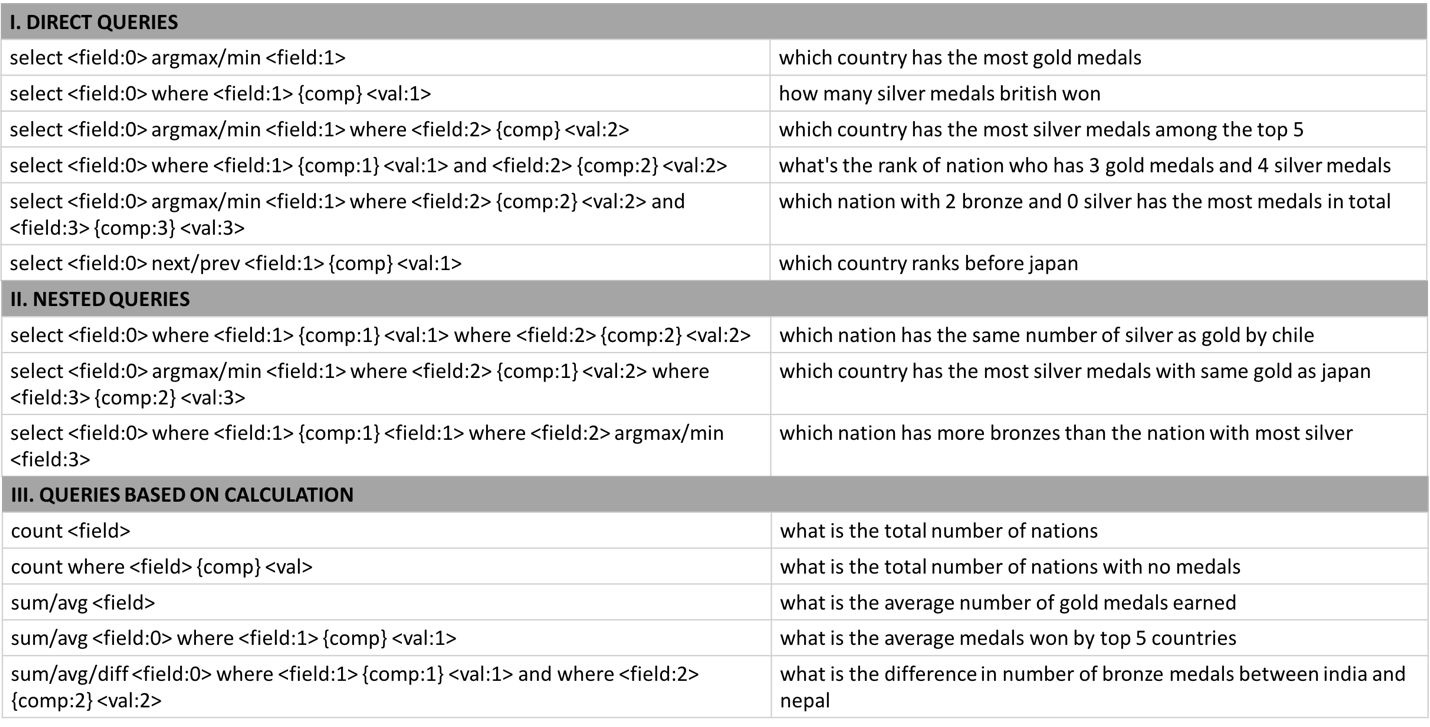


The figure above shows a detailed example of the query optimization process, tagging an input query (‘.qu’ file) and recording the information of field and value appeared (‘.ficorr’ and ‘.vacorr’ files); one information to note is the ‘.qux’ file shows the hybrid between the original query and its tag, where the tokens is replaced by its label if the label is not ‘<nan>’. In fact, we have two different options when feeding the optimized query into the NLI: one way is sending both original query and the tags in parallel, denoted as the ‘Parallel mode’; the other way is to feed in the hybrid query described earlier, denoted as the ‘X mode’.

As a side note, the query optimization process significantly reduces the effect of words unseen during the training process; since we can still tag them with possible field names or values, and then use dependency-tree to establish a correspondence. As long as the optimization stage accurately tags the query, the NLI inference process will still generate the correct logical template.

3.2 Declarative command

We choose to parse the natural language query to declarative command like SQL, so the database could be directly accessed. From the example shown above, we can see the correspondence between a query (‘.qu’ file) and a SQL command (‘.lo’ file). To train a general NLI, we use placeholder to take the places of actual field names and values to create a logical template (‘.lox’ file), and use the SQL logical template as the decoder input; the token of placeholder is the same as the tag used to label the query. Then during the inference process, after NLI generates a logical template, we could use the field-value correspondence information saved during the query optimization stage to recover an actual SQL command.



3.3 NLIDB: Seq2seqBridge model

(new figure of Seq2seqBridge, with tiny side figure showing tag features)

To convert the input tagged query (either Parallel mode or X mode) to SQL logical template, we designed an improved deep neural network based on the vanilla attentive seq2seq network, enabling the information sharing and synchronic updating between encoder and decoder.

In vanilla attentive seq2seq network, (based on original written)

The vanilla seq2seq works very well in machine translation problem, where inputs and outputs are using completely different vocabulary and word embedding. However, for our NLI, the encoder side and decoder side contains same information regarding the tagging. If we make the placeholders appeared in SQL logical template and the tags of optimized query share the same embedding, and synchronize their updates during training, it will greatly improve the semantic parsing ability of the NLI. We will explain in detail about how we design features for the tag/placeholder, and how we realize the embedding sharing and synchronic updating.

3.2.1 Features of tag

The simplest way is to use a separate embedding for tag/placeholder, assigning different vectors for different tags. However, the simplest method does not consider the fact that ‘<field:0>’ and ‘<field:1>’ are both field indicators and should have some similarities, and so as that ‘<field:0>’ and ‘<value:0>’ are both indicators for the 0th field, and should also have similarities.

We come up with a better way to assign embedding to tags with shared features, instead of using totally different vectors. From the figure, we see the embedding for each tag composes several segments: the first segment represents either ‘<field>’ or ‘<value>’, and the second segment represents the id (later we further incorporate the value type information as the third segment into the tag feature). In this way, similar tags would share part of the embeddings. For example, in our experiment, the embedding features for each tag is with dimension of 100, which is concatenated by a 50-dimension vector representing the category and another 50-dimension vector representing the id.

3.2.2 Seq2seqBridge model

We design the Seq2seqBridge model to enable the embedding sharing of tag/placeholder between encoder and decoder. Assume the dimension for the encoder embedding and decoder embedding is D, then we generate another feature embedding with dimension D/2 for the tag. For each tag, the representations for ‘<field>’, ‘<value>’, and id could be looked up in the feature embedding matrix, and then the vector for ‘<field>’ or ‘<value>’ and the vector for id are concatenated together to be the embedding of the tag. The same feature embedding is used for both tag in the tagged query at encoder side, and the placeholder in SQL logical template at decoder side, so the corresponding vectors at both sides will be updated synchronically during training. The feature embedding for the tag/placeholder serves as a bridge sharing information and synchronizing updates between encoder and decoder, and enables the NLI to better capture the correspondence between the structure of query and the structure of SQL command, rather than just the correspondence between a query and a logical form.

Previous works have explored pointer network [] and attention-based copying [] to deal with words that appear in both encoder side and decoder side, with the help of attention mechanism, which is different from the Seq2seqBridge model we are using here. Pointer Network [] is based on attention mechanism to select one exact state at encoder side which has the highest attention score, and the vocabulary at decoder side is the same as the encoder side; the attention-based copying [] works similarly, while the vocabulary of decoder side is the original translate-to-vocabulary combined with the translate-from-vocabulary at encoder side. Although Seq2seqBridge also has the attention mechanism, it does not depend on the attention score to help connect between the same tokens in encoder and decoder; instead, it uses a feature embedding matrix for the tag/placeholder, which offers shared information and synchronic updating on both sides, thus establishing a more direct and stable way to bridge the same vocabulary on both sides

4 Cross-domain transfer-learning protocol

4.1 Transfer-learning protocol

(figure)

When we are given a new database, how do we transfer the pre-trained NLI model to this different knowledge domain? Even for databases quite different from each other, there are some general and shared rules that you can make queries on. For example, we can ask “what is the rank of us” in Wikitable, and similarly we can ask “what is the population of alabama”; the NLI is able to parse this kind of queries directly on the different database, because the query optimization process can successfully tag the query and the NLI is able to recognize the structure of the tagged query and infer the correct SQL logical template. Thus, there is no need to re-train the NLI model from scratch.

However, we should notice that the new database could have very different domains and predicates than the ones the NLI has been trained on; for example, in addition to asking “what is the population of alabama”, we can also ask “how populous is alabama” to get the same SQL command, but the variant query has not necessarily been seen when the NLI is trained on former databases. The new database could even have different relational schema which involves some different SQL logical templates that the pre-trained NLI has not seen. For example, if we have trained a NLI on Wikitable, and consider transferring the model to the GeoQuery database, there are certain queries which are very difficult to be parsed by the pre-trained NLI, such as “which state has the largest population run through by mississippi river”, because the entities ‘state’ and ‘river’ in GeoQuery have more complex graph-like relationships than the entities in Wikitable. Therefore, we should not directly apply the pre-trained NLI on a new database, but need fine-tuned training to educate the NLI with knowledge regarding the new database.

Therefore, we have come up with a transfer-learning protocol: (1) Given some prior knowledge of the schema (field names and their value domains), and several queries examples specific to this database, (2) we can augment a small dataset and then (3) use the augmented dataset to fine-tune the NLI model. After the fine-tuned training, we could infer on queries related to new database.

4.2 Data augmentation ([Ref. Robin Jia])

Based on the transfer-learning protocol, an effective data augmentation technique is essential for the fine-tuned training process. Typically, the queries examples are a tiny portion of the original dataset specifically chosen to cover the unique query types that will be involved in this database (not general or shared types with other databases). For each query example provided, we can label it with the correct SQL command; then given the schema information, we can generate the hybrid query (‘.qux’ file) and the SQL logical template (‘.lox’) using the tagging algorithm in query optimization process

(Mostly original written)

5 Experiment

5.1 Dataset

Wikitable --- a new dataset, data augmentation based on Wikitable

GeoQuery880 ---

Overnight ---

; cross-domain transfer-learnable NLIDB, Geo880 (Overnight), the SQL logical template could be deterministically generated from the lambda expression.

5.2 Seq2seqBridge model on Wikitable, GeoQuery and Overnight, respectively

(Table)

The reason that Parallel mode has a better performance is that X mode tends to over-simplify the information and complexity of the input query. By replacing the tokens in original query by their corresponding tags, it is possible that the replacement will delete important information contained in the query. For example, the query “which river is longest”, where the token ‘longest’ represents the field ‘River’ and simultaneously indicate the information of ‘argmax’ operation, so replacing this token with its tag will drop a very important piece of knowledge from the original query.

5.3 Transfer learning on GeoQuery and Overnight

More than 70% of the words in GeoQuery never occur in Wikitable; what is more, in addition to different predicates, there are also different relational structure in GeoQuery which has not been seen during the training on Wikitable.

6 Conclusion

We develop a general Transfer-learnable Deep Neural NLIDB Pipeline, which processes the input queries to convert to SQL-extended commands, which could be used to access DBMS;

The tagging algorithm gives a decent query optimization, adding the schema information to the original query;

The Seq2seqBridge model enables information communication between the tagged query and corresponding SQL-extended command, giving a higher test accuracy than normal seq2seq model;

Data augmentation boosts the volume of dataset while preserve the complexity;

We finally establish a database-adaptation protocol which transfers the formally-trained NLI on unseen different database with high accuracy.