Editing-Based SQL Query Generation for Cross-Domain Context-Dependent Questions

Rui Zhang† Tao Yu† He Yang Er† Sungrok Shim† Eric Xue† Xi Victoria Lin‡ Tianze Shi§ Caiming Xiong¶ Richard Socher¶ Dragomir Radev†
† Yale University ‡ Salesforce Research § Cornell University
{r.zhang, tao.yu, dragomir.radev}@yale.edu
{xilin, cxiong, rsocher}@salesforce.com

Abstract

We focus on the cross-domain context-dependent text-to-SQL generation task. Based on the observation that adjacent natural language questions are often linguistically dependent and their corresponding SQL queries tend to overlap, we utilize the interaction history by editing the previous predicted query to improve the generation quality. Our editing mechanism views SQL as sequences and reuses generation results at the token level in a simple manner. It is flexible to change individual tokens and robust to error propagation. Furthermore, to deal with complex table structures in different domains, we employ an utterance-table encoder and a table-aware decoder to incorporate the context of the user utterance and the table schema. We evaluate our approach on the SParC dataset and demonstrate the benefit of editing compared with the state-of-the-art baselines which generate SQL from scratch. Our code is available at https://github.com/ryanzhumich/sparc_atis_pytorch.

1 Introduction

Generating SQL queries from user utterances is an important task to help end users acquire information from databases. In a real-world application, users often access information in a multi-turn interaction with the system by asking a sequence of related questions. As the interaction proceeds, the user often makes reference to the relevant mentions in the history or omits previously conveyed information assuming it is known to the system.

Therefore, in the context-dependent scenario, the contextual history is crucial to understand the follow-up questions from users, and the system often needs to reproduce partial sequences generated in previous turns. Recently, Suhr et al. (2018) propose a context-dependent text-to-SQL model including an interaction-level encoder and an attention mechanism over previous utterances. To reuse what has been generated, they propose to copy complete segments from the previous query. While their model is successful to reason about explicit and implicit references, it does not need to explicitly address different database schemas because the ATIS contains only the flight-booking domain. Furthermore, the model is confined to copy whole segments which are extracted by a rule-based procedure, limiting its capacity to utilize the previous query when only one or a few tokens are changed in the segment.

To exploit the correlation between sequentially generated queries and generalize the system to different domains, in this paper, we study an editing-based approach for cross-domain context-dependent text-to-SQL generation task. We propose query generation by editing the query in the previous turn. To this end, we first encode the previous query as a sequence of tokens, and the decoder computes a switch to change it at the token level. This sequence editing mechanism models token-level changes and is thus robust to error propagation. Furthermore, to capture the user utterance and the complex database schemas in different domains, we use an utterance-table encoder based on BERT to jointly encode the user utterance and column headers with co-attention, and adopt a table-aware decoder to perform SQL generation with attentions over both the user utterance and column headers.

We evaluate our model on SParC (Yu et al., 2019b), a new large-scale dataset for cross-domain semantic parsing in context consisting of coherent question sequences annotated with SQL queries over 200 databases in 138 domains. Experiment results show that by generating from the previous query, our model delivers an improvement of 7% question match accuracy and 11% interaction...
Table 1: Dataset Statistics.

<table>
<thead>
<tr>
<th>Database</th>
<th>Interaction (train / dev / test)</th>
<th>Question</th>
<th>Turn</th>
<th>Database</th>
<th>Table</th>
<th>Q. Length</th>
<th>Q. Vocab</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spider</td>
<td>✓</td>
<td>11,840 (8,659 / 1,034 / 2,147)</td>
<td>11,840</td>
<td>1.0</td>
<td>200</td>
<td>1020</td>
<td>13.4</td>
</tr>
<tr>
<td>ATIS</td>
<td>✓</td>
<td>1,658 (1,148 / 380 / 130)</td>
<td>11,653</td>
<td>7.0</td>
<td>1</td>
<td>27</td>
<td>10.2</td>
</tr>
<tr>
<td>SParC</td>
<td>✓ ✓</td>
<td>4,298 (3,034 / 422 / 842)</td>
<td>12,726</td>
<td>3.0</td>
<td>200</td>
<td>1020</td>
<td>8.1</td>
</tr>
</tbody>
</table>

Table 2: % of SQL queries that contain a particular SQL component.

<table>
<thead>
<tr>
<th>SQL Component</th>
<th>Spider</th>
<th>ATIS</th>
<th>SParC</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHERE</td>
<td>55.2</td>
<td>100</td>
<td>42.8</td>
</tr>
<tr>
<td>AGG</td>
<td>51.7</td>
<td>16.6</td>
<td>39.8</td>
</tr>
<tr>
<td>GROUP</td>
<td>24.0</td>
<td>0.3</td>
<td>20.1</td>
</tr>
<tr>
<td>ORDER</td>
<td>21.5</td>
<td>0</td>
<td>17.0</td>
</tr>
<tr>
<td>HAVING</td>
<td>6.7</td>
<td>0</td>
<td>4.7</td>
</tr>
<tr>
<td>SET</td>
<td>5.8</td>
<td>0</td>
<td>3.5</td>
</tr>
<tr>
<td>JOIN</td>
<td>42.9</td>
<td>96.6</td>
<td>35.5</td>
</tr>
<tr>
<td>Nested</td>
<td>15.7</td>
<td></td>
<td>5.7</td>
</tr>
</tbody>
</table>

Database: student dormitory containing 5 tables.
Goal: Find the first and last names of the students who are living in the dorms that have a TV Lounge as an amenity.
Q1: How many dorms have a TV Lounge?
S1: SELECT COUNT(*) FROM dorm AS T1 JOIN has_amenity AS T2 ON T1.dormid = T2.dormid JOIN dorm_amenity AS T3 ON T2.amenid = T3.amenid WHERE T3.amenity_name = 'TV Lounge'
Q2: What is the total capacity of these dorms?
S2: SELECT SUM(T1.student_capacity) FROM dorm AS T1 JOIN has_amenity AS T2 ON T1.dormid = T2.dormid JOIN dorm_amenity AS T3 ON T2.amenid = T3.amenid WHERE T3.amenity_name = 'TV Lounge'
Q3: How many students are living there?
S3: SELECT COUNT(*) FROM student AS T1 JOIN lives_in AS T2 ON T1.stuid = T2.stuid WHERE T2.dormid IN (SELECT T3.dormid FROM has_amenity AS T3 JOIN dorm_amenity AS T4 ON T3.amenid = T4.amenid WHERE T4.amenity_name = 'TV Lounge')
Q4: Please show their first and last names.
S4: SELECT T1.fname, T1.lname FROM student AS T1 JOIN lives_in AS T2 ON T1.stuid = T2.stuid WHERE T2.dormid IN (SELECT T3.dormid FROM has_amenity AS T3 JOIN dorm_amenity AS T4 ON T3.amenid = T4.amenid WHERE T4.amenity_name = 'TV Lounge')

Table 3: SParC example.

match accuracy over the previous state-of-the-art. Further analysis shows that our editing approach is more robust to error propagation than copying segments, and the improvement becomes more significant if the basic text-to-SQL generation accuracy (without editing) improves.

2 Cross-Domain Context-Dependent Semantic Parsing

2.1 Datasets

We use SParC 1 (Yu et al., 2019b), a large-scale cross-domain context-dependent semantic parsing dataset with SQL labels, as our main evaluation benchmark. A SParC example is shown in Table 3. We also report performance on ATIS (Hemphill et al., 1990; Dahl et al., 1994a) for direct comparison to Suhr et al. (2018). In addition, we evaluate the cross-domain context-independent text-to-SQL ability of our model on Spider 2 (Yu et al., 2018c), which SParC is built on.

We summarize and compare the data statistics in Table 1 and Table 2. While the ATIS dataset has been extensively studied, it is limited to a particular domain. By contrast, SParC is both context-dependent and cross-domain. Each interaction in SParC is constructed using a question in Spider as the interaction goal, where the annotator asks inter-related questions to obtain information that completes the goal. SParC contains interactions over 200 databases and it follows the same database split as Spider where each database appears only in one of train, dev and test sets. In summary, SParC introduces new challenges to context-dependent text-to-SQL because it (1) contains more complex context dependencies, (2) has greater semantic coverage, and (3) adopts a cross-domain task setting.

2.2 Task Formulation

Let X denote a natural language utterance and Y denote the corresponding SQL query. Context-

---

1https://yale-lily.github.io/sparc
2https://yale-lily.github.io/spider
What is the customer id of the most recent customer?
What is their name?

SELECT customer_id FROM customers ORDER BY date_became_customer DESC LIMIT 1

LSTM Query Decoder

Interaction Encoder

Utterance Encoder

SELECT customer_name FROM customers ORDER BY date_became_customer DESC LIMIT 1

LSTM Query Decoder

Attention Over Previous Utterances, Column Headers, Previous Query

Bi LSTM Query Encoder

Table Encoder

Table: customers

| customer_id | customer_name |

Figure 1: Model architecture of editing the previous query with attentions to the user utterances, the table schema, and the previously generated query.

独立的语义解析考虑了个体 \((X, Y)\) 对，并映射 \(X\) 到 \(Y\)。在基于上下文的语义解析中，我们考虑一个由 \(n\) 个对话-查询对组成的互动：

\[ I = [(X_i, Y_i)]_{i=1}^n \]

在每次轮换 \(t\)，目标是根据当前的对话 \(X_t\) 和互动历史

\[ [(X_1, Y_1), (X_2, Y_2), \ldots, (X_{t-1}, Y_{t-1})] \]

此外，在跨域设置中，每次互动都基于不同的数据库。因此，该模型也给出了当前数据库的结构输入。我们考虑具有多个表的事务性数据库，每个表包含多个列标题：

\[ T = [c_1, c_2, \ldots, c_l, \ldots, c_m] \]

其中 \(m\) 是列标题的数量，而每个 \(c_l\) 包含表名和列名（§ 3.1）。

3 Methodology

我们采用编码器-解码器架构并利用注意机制（Sutskever et al., 2014; Luong et al., 2015）如图1所示。该框架由（1）一个对话-表编码器，用来解析用户的对话和表结构，（2）一个表结构的注意力，来捕捉对话的上下文和表结构的上下文，（3）一个可传递的注意力，注意力取决于用户的对话和表结构的上下文，（4）一个第二层双向LSTM来生成列标题嵌入 \(h^C\)。

图2b显示了对话编码器。对于每次对话，我们首先使用双向LSTM来编码句子词向量。双向LSTM的隐藏状态被馈入一个点乘注意层（Luong et al., 2015）来对列标题进行加权平均。对于每个对话词向量，我们得到一个列标题的注意力加权平均来获得最相关的列（Dong and Lapata, 2018）。然后，我们拼接双向LSTM的隐藏状态和列标题的注意力向量，再用第二层双向LSTM来生成对话词向量。

注意，这两个嵌入由于协同注意依赖于彼此，列标题的表示会随对话变化。
Utterance-Table BERT Embedding. We consider two options as the input to the first layer bi-LSTM. The first choice is the pretrained word embedding. Second, we also consider the contextualized word embedding based on BERT (Devlin et al., 2019). To be specific, we follow Hwang et al. (2019) to concatenate the user utterance and all the column headers in a single sequence separated by the [SEP] token:

$$[CLS], X_i, [SEP], c_1, [SEP], ..., c_m, [SEP]$$

This sequence is fed into the pretrained BERT model whose hidden states at the last layer is used as the input embedding.

3.2 Interaction Encoder with Turn Attention

To capture the information across different utterances, we use an interaction-level encoder (Suhr et al., 2018) on top of the utterance-level encoder. At each turn, we use the hidden state at the last time step from the utterance-level encoder as the utterance encoding. This is the input to a unidirectional LSTM interaction encoder:

$$h_t^U = h_E^U x_t$$

$$h_{t+1} = \text{LSTM}'(h_t^U, h_t')$$

The hidden state of this interaction encoder $h_t^I$ encodes the history as the interaction proceeds.

**Turn Attention** When issuing the current utterance, the user may omit or explicitly refer to the previously mentioned information. To this end, we adopt the turn attention mechanism to capture correlation between the current utterance and the utterance(s) at specific turn(s). At the current turn $t$, we compute the turn attention by the dot-product attention between the current utterance and previous utterances in the history, and then add the weighted average of previous utterance embeddings to the current utterance embedding:

$$s_t = h_t^U W_{\text{turn-att}} h_t^U$$

$$\alpha_{\text{turn}} = \text{softmax}(s)$$

$$c_t = h_t^U + \sum_{i=1}^{t-1} \alpha_{\text{turn}}^i \times h_i^U$$

The $c_t$ summarizes the context information and the current user query and will be used as the initial decoder state as described in the following.

3.3 Table-aware Decoder

We use an LSTM decoder with attention to generate SQL queries by incorporating the interaction
history, the current user utterance, and the table schema.

Denote the decoding step as $k$, we provide the decoder input as a concatenation of the embedding of SQL query token $q_k$ and a context vector $c_k$:

$$h_{k+1}^D = \text{LSTM}^D([q_k; c_k], h_k^D)$$

where $h^D$ is the hidden state of the decoder LSTM$^D$, and the hidden state $h_0^D$ is initialized by $c_{\text{turn}}$. When the query token is a SQL keyword, $q_k$ is a learned embedding; when it is a column header, we use the column header embedding given by the table-utterance encoder as $q_k$. The context vector $c_k$ is described below.

**Context Vector with the Table and User Utterance.** The context vector consists of attentions to both the table and the user utterance. First, at each step $k$, the decoder computes the attention between the decoder hidden state and the column header embedding:

$$s_l = h_k^D W_{\text{column-att}} h_l^C$$

$$\alpha_{\text{column}} = \text{softmax}(s)$$

$$c_k^{\text{column}} = \sum_l \alpha_l^{\text{column}} \times h_l^C$$

where $l$ is the index of column headers and $h_l^C$ is its embedding. Second, it also computes the attention between the decoder hidden state and the utterance token embeddings:

$$s_{i,j} = h_k^D W_{\text{utterance-att}} h_{i,j}^E$$

$$\alpha_{\text{utterance}} = \text{softmax}(s)$$

$$c_k^{\text{token}} = \sum_{i,j} \alpha_{i,j}^{\text{utterance}} \times h_{i,j}^E$$

where $i$ is the turn index, $j$ is the token index, and $h_{i,j}^E$ is the token embedding for the $j$-th token of the $i$-th utterance. The context vector $c_k$ is a concatenation of the two:

$$c_k = [c_k^{\text{column}}; c_k^{\text{token}}]$$

**Output Distribution.** In the output layer, our decoder chooses to generate a SQL keyword (e.g., `SELECT`, `WHERE`, `GROUP BY`, `ORDER BY`) or a column header. This is critical for the cross-domain setting where the table schema changes across different examples. To achieve this, we use separate layers to score SQL keywords and column headers, and finally use the softmax operation to generate the output probability distribution:

$$o_k = \tanh([h_k^D; c_k] W_o)$$

$$m_{\text{SQL}} = o_k W_{\text{SQL}} + b_{\text{SQL}}$$

$$m_{\text{column}} = o_k W_{\text{column}} h^C$$

$$P(y_k) = \text{softmax}([m_{\text{SQL}}; m_{\text{column}}])$$

### 3.4 Query Editing Mechanism

In an interaction with the system, the user often asks a sequence of closely related questions to complete the final query goal. Therefore, the query generated for the current turn often overlaps significantly with the previous ones.

To empirically verify the usefulness of leveraging the previous query, we consider the process of generating the current query by applying copy and insert operations to the previous query. Figure 3 shows the SQL query length and the number of copy and insert operations at different turns. As the interaction proceeds, the user question becomes more complicated as it requires longer SQL query to answer. However, more query tokens overlap with the previous query, and thus the number of new tokens remains small at the third turn and beyond.

Based on this observation, we extend our tableware decoder with a query editing mechanism. We first encode the previous query using another bi-LSTM, and its hidden states are the query token embeddings $h_{i,j}^Q$ (i.e., the $j$-th token of the $i$-th query). We then extend the context vector with the attention to the previous query:

$$c_k = [c_k^{\text{column}}; c_k^{\text{token}}; c_k^{\text{query}}]$$

where $c_k^{\text{query}}$ is produced by an attention to query tokens $h_{i,j}^Q$ in the same form as Equation 3.

---

3We use a diffing algorithm from [https://github.com/paulgb/simplesdiff](https://github.com/paulgb/simplesdiff)
At each decoding step, we predict a switch \( p_{\text{copy}} \) to decide if we need copy from the previous query or insert a new token.

\[
p_{\text{copy}} = \sigma(c_k W_{\text{copy}} + b_{\text{copy}}) \\
p_{\text{insert}} = 1 - p_{\text{copy}}
\]  

(5)

Then, we use a separate layer to score the query tokens at turn \( t - 1 \), and the output distribution is modified as the following to take into account the editing probability:

\[
P_{\text{prev,SQL}} = \text{softmax}(o_h W_{\text{prev,SQL}} h^Q_{t-1}) \\
m^{\text{SQL}} = o_h W_{\text{SQL}} + b_{\text{SQL}} \\
m^{\text{column}} = o_h W_{\text{column}} h^C \\
P_{\text{SQL} \cup \text{column}} = \text{softmax}([m^{\text{SQL}}; m^{\text{column}}]) \\
P(y_k) = p_{\text{copy}} \cdot P_{\text{prev,SQL}}(y_k \in \text{prev,SQL}) + p_{\text{insert}} \cdot P_{\text{SQL} \cup \text{column}}(y_k \in \text{SQL} \cup \text{column})
\]  

(6)

While the copy mechanism has been introduced by Gu et al. (2016) and See et al. (2017), they focus on summarization or response generation applications by copying from the source sentences. By contrast, our focus is on editing the previously generated query while incorporating the context of user utterances and table schemas.

4 Related Work

Semantic parsing is the task of mapping natural language sentences into formal representations. It has been studied for decades including using linguistically-motivated compositional representations, such as logical forms (Zelle and Mooney, 1996; Clarke et al., 2010) and lambda calculus (Zettlemoyer and Collins, 2005; Artzi and Zettlemoyer, 2011), and using executable programs, such as SQL queries (Miller et al., 1996; Zhong et al., 2017) and other general-purpose programming languages (Yin and Neubig, 2017; Iyer et al., 2018). Most of the early studies worked on a few domains and small datasets such as GeoQuery (Zelle and Mooney, 1996) and Overnight (Wang et al., 2015).

Recently, large and cross-domain text-to-SQL datasets such as WikiSQL (Zhong et al., 2017) and Spider (Yu et al., 2018c) have received an increasing amount of attention as many data-driven neural approaches achieve promising results (Dong and Lapata, 2016; Su and Yan, 2017; Iyer et al., 2017; Xu et al., 2017; Finegan-Dollak et al., 2018; Yu et al., 2018a; Huang et al., 2018; Dong and Lapata, 2018; Sun et al., 2018; Gur et al., 2018; Guo et al., 2018; Yavuz et al., 2018; Shi et al., 2018).

Most of them still focus on context-independent semantic parsing by converting single-turn questions into executable queries.

Relatively less effort has been devoted to context-dependent semantic parsing on datasets including ATIS (Hemphill et al., 1990; Dahl et al., 1994b), SpaceBook (Vlachos and Clark, 2014), SCONE (Long et al., 2016; Guu et al., 2017; Fried et al., 2018; Suhr and Artzi, 2018; Huang et al., 2019), SequentialQA (Iyyer et al., 2017), SParC (Yu et al., 2019b) and CoSQL (Yu et al., 2019a).

On ATIS, Miller et al. (1996) maps utterances to semantic frames which are then mapped to SQL queries; Zettlemoyer and Collins (2009) starts with context-independent Combinatory Categorial Grammar (CCG) parsing and then resolves references to generate lambda-calculus logical forms for sequences of sentences. The most relevant to our work is Suhr et al. (2018), who generate ATIS SQL queries from interactions by incorporating history with an interaction-level encoder and copying segments of previously generated queries. Furthermore, SCONE contains three domains using stack- or list-like elements and most queries include a single binary predicate. SequentialQA is created by decomposing some complicated questions in WikiTableQuestions (Passpat and Liang, 2015). Since both SCONE and SequentialQA are annotated with only denotations but not query labels, they don’t include many questions with rich semantic and contextual types. For example, SequentialQA (Iyyer et al., 2017) requires that the answer to follow-up questions must be a subset of previous answers, and most of the questions can be answered by simple SQL queries with SELECT and WHERE clauses.

Concurrent with our work, Yu et al. (2019a) introduced CoSQL, a large-scale cross-domain conversational text-to-SQL corpus collected under the Wizard-of-Oz setting. Each dialogue in CoSQL simulates a DB querying scenario with a crowd worker as a user and a college computer science student who is familiar with SQL as an expert. Question-SQL pairs in CoSQL reflect greater diversity in user backgrounds compared to other corpora and involve frequent changes in user intent between pairs or ambiguous questions that require user clarification. These features pose new chal-
challenges for text-to-SQL systems.

Our work is also related to recently proposed approaches to code generation by editing (Hayati et al., 2018; Yin et al., 2019; Hashimoto et al., 2018). While they follow the framework of generating code by editing the relevant examples retrieved from training data, we focus on a context-dependent setting where we generate queries from the previous query predicted by the system itself.

5 Experimental Results

5.1 Metrics

On both Spider and SParC, we use the exact set match accuracy between the gold and the predicted queries. To avoid ordering issues, instead of using simple string matching, Yu et al. (2018c) decompose predicted queries into different SQL clauses such as SELECT, WHERE, GROUP BY, and ORDER BY and compute scores for each clause using set matching separately. On SparC, we report two metrics: question match accuracy which is the score average over all questions and interaction match accuracy which is average over all interactions.

5.2 Baselines

SParC. We compare with the two baseline models released by Yu et al. (2019b).

(1) Context-dependent Seq2Seq (CD-Seq2Seq): This model is adapted from Suhr et al. (2018). The original model was developed for ATIS and does not take the database schema as input hence cannot generalize well across domains. Yu et al. (2019b) adapt it to perform context-dependent SQL generation in multiple domains by adding a bi-LSTM database schema encoder which takes bag-of-words representations of column headers as input. They also modify the decoder to select between a SQL keyword or a column header.

(2) SyntaxSQL-con: This is adapted from the original context-agnostic SyntaxSQLNet (Yu et al., 2018b) by using bi-LSTMs to encode the interaction history including the utterance and the associated SQL query response. It also employs a column attention mechanism to compute representations of the previous question and SQL query.

Spider. We compare with the results as reported in Yu et al. (2018b). Furthermore, we also include recent results from Lee (2019) who propose to use recursive decoding procedure, Bogin et al. (2019) introducing graph neural networks for encoding schemas, and Guo et al. (2019) who achieve state-of-the-art performance by using an intermediate representation to bridge natural language questions and SQL queries.

5.3 Implementation Details

Our model is implemented in PyTorch (Paszke et al., 2017). We use pretrained 300-dimensional GloVe (Pennington et al., 2014) word embedding. All LSTM layers have 300 hidden size, and we use 1 layer for encoder LSTMs, and 2 layers for decoder LSTMs. We use the ADAM optimizer (Kingma and Ba, 2015) to minimize the token-level cross-entropy loss with a batch size of 16. Model parameters are randomly initialized from a uniform distribution $U[-0.1, 0.1]$. The main model has an initial learning rate of 0.001 and it will be multiplied by 0.8 if the validation loss increases compared with the previous epoch. When using BERT instead of GloVe, we use the pre-trained small uncased BERT model with 768 hidden size, and we fine tune it with a separate constant learning rate of 0.00001. The training typically converges in 10 epochs.

5.4 Overall Results

Spider. Table 4 shows the results on Spider dataset. Since each question is standalone, we don’t use interaction-level decoder or query editing. Our method can achieve the performance of 36.4% on dev set and 32.9% on test set, serving as a strong model for the context-independent cross-domain text-to-SQL generation. This demonstrates the effectiveness of our utterance-table encoder and table-aware decoder to handle the semantics of user utterances and the complexity of table schemas to generate complex SQL queries in unseen domains.

Furthermore, adding the utterance-table BERT

\[ \text{Table 4: Spider results on dev set and test set.} \]

<table>
<thead>
<tr>
<th>Model</th>
<th>Dev Set</th>
<th>Test Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLNet (Xu et al., 2017)</td>
<td>10.9</td>
<td>12.4</td>
</tr>
<tr>
<td>SyntaxSQLNet (Yu et al., 2018b)</td>
<td>18.9</td>
<td>19.7</td>
</tr>
<tr>
<td>+ data augmentation (Yu et al., 2018b)</td>
<td>24.8</td>
<td>27.2</td>
</tr>
<tr>
<td>Lee (2019)</td>
<td>28.5</td>
<td>24.3</td>
</tr>
<tr>
<td>GNN (Bogin et al., 2019)</td>
<td>40.7</td>
<td>39.4</td>
</tr>
<tr>
<td>IRNet (Guo et al., 2019)</td>
<td>53.2</td>
<td>46.7</td>
</tr>
<tr>
<td>IRNet (BERT) (Guo et al., 2019)</td>
<td>61.9</td>
<td>54.7</td>
</tr>
<tr>
<td>Ours</td>
<td>36.4</td>
<td>32.9</td>
</tr>
<tr>
<td>+ utterance-table BERT Embedding</td>
<td>57.6</td>
<td>53.4</td>
</tr>
</tbody>
</table>

4More details at https://github.com/taoyds/spider/tree/master/evaluation_examples

5https://github.com/google-research/bert
embedding gives significant improvement, achieving 57.6% on dev set and 53.4% on test set, which is comparable to the state-of-the-art results from IRNet with BERT. We attribute our BERT model’s high performance to (1) the empirically powerful text understanding ability of pretrained BERT model and (2) the early interaction between utterances and column headers when they are concatenated in a single sequence as the BERT input.

**SParC.** Table 5 shows the results on SParC dataset. Similar to Spider, our model without previous query as input already outperforms SyntaxSQL-con, achieving 31.4% question matching accuracy and 14.7% interaction matching accuracy. In addition, compared with CD-Seq2Seq, our model enjoys the benefits of the table-utterance encoder, turn attention, and the joint consideration of utterances and table schemas during the decoding stage. This boosts the performance by 10% question accuracy and 6% interaction accuracy.

Furthermore, we also investigate the effect of copying segment. We use the same segment copy procedure as Suhr et al. (2018): first deterministically extract segments from the previous query and encode each segment using an LSTM, then generate a segment by computing its output probability based on its segment encoding. However, since the segment extraction from Suhr et al. (2018) is exclusively designed for the ATIS dataset, we implement our own segment extraction procedure by extracting SELECT, FROM, GROUP BY, ORDER BY clauses as well as different conditions in WHERE clauses. In this way, 3.9 segments can be extracted per SQL on average. We found that adding segment copying to CD-Seq2Seq gives a slightly lower performance on question matching and a small gain on interaction matching, while using segments extracted from the gold query can have much higher results. This demonstrates that segment copy is vulnerable to error propagation. In addition, it can only copy whole segments hence has difficulty capturing the changes of only one or a few tokens in the query.

To better understand how models perform as the interaction proceeds, Figure 4 (Left) shows the performance split by turns on the dev set. The questions asked in later turns are more difficult to answer given longer context history. While the baselines have lower performance as the turn number increases, our model still maintains 38%-48% accuracy for turn 2 and 3, and 20% at turn 4 or beyond. Similarly, Figure 4 (Right) shows the performance split by hardness levels with the frequency of examples. This also demonstrates our model is more competitive in answering hard and extra hard questions.

**ATIS.** We also report our model performance
on ATIS in Table 6. Our model achieves 36.2% dev and 43.9% test string accuracy, comparable to Suhr et al. (2018). On ATIS, we only apply our editing mechanism and reuse their utterance encoder instead of the BERT utterance-table encoder, because ATIS is single domain.

5.5 Effect of Query Editing

We further investigate the effect of our query editing mechanism. To this end, we apply editing from both the gold query and the predicted query on our model with or without the utterance-table BERT embedding. We also perform an ablation study to validate the contribution of query attention and sequence editing separately.

As shown in Table 5, editing the gold query consistently improves both question match and interaction match accuracy. This shows the editing approach is indeed helpful to improve the generation quality when the previous query is the oracle.

Using the predicted query is a more realistic setting, and in this case, the model is affected by error propagation due to the incorrect queries produced by itself. For the model without the utterance-table BERT embedding, using the predicted query only gives around 1.5% improvement. As shown in Figure 5, this is because the editing mechanism is more helpful for turn 4 which is a small fraction of all question examples. For the model with the utterance-table BERT embedding, the query generation accuracy at each turn is significantly improved, thus reducing the error propagation effect. In this case, the editing approach delivers consistent improvements of 7% increase on question matching accuracy and 11% increase on interaction matching accuracy. Figure 5 also shows that query editing with BERT benefits all turns.

Finally, as an ablation study, Table 5 also reports the result with only query attention (use predicted query) on the dev set. This improves over our vanilla BERT model without query attention and achieves 42.7% question and 21.6% interaction matching accuracy. With query editing, our best model further improves to 47.2% question and 29.5% interaction matching accuracy. This demonstrates the effectiveness of our query attention and query editing separately, both of which are essential to make use of the previous query.

6 Conclusions

In this paper, we propose an editing-based encoder-decoder model to address the problem of context-dependent cross-domain text-to-SQL generation. While being simple, empirical results demonstrate the benefits of our editing mechanism. The approach is more robust to error propagation than copying segments, and its performance increases when the basic text-to-SQL generation quality (without editing) is better.

Acknowledgements

We thank the anonymous reviewers for their thoughtful detailed comments.
References


Minh-Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective approaches to attention-based neural machine translation. In EMNLP.


