A Syntax-Guided Neural Model for Natural Language Interfaces to Databases

Florin Brad[1], Radu Iacob[2], Ionel Hosu[2], Stefan Ruseti[2], and Traian Rebedea[2]

[1]Bitdefender Romania
fbrad@bitdefender.com
[2]University Politehnica of Bucharest
{*radu.iacob, ionel.hosu, stefan.ruseti, traian.rebedea*}*@cs.pub.ro*

***Abstract*—Recent advances in neural code generation have incorporated syntax to improve the generation of the target code based on the user's request in natural language. We adapt the model of [1] to the Natural Language Interface to Databases (NLIDB) problem by taking into account the database schema. We evaluate our model on the recently introduced WIKISQL and SENLIDB datasets. Our results show that the syntax-guided model outperforms a simple sequence-to-sequence (SEQ2SEQ) baseline on WIKISQL, but has trouble with the SENLIDB dataset due to its complexity.**

I. INTRODUCTION

Endowing non-technical users with the ability to investigate large amounts of data is a practical AI tool that could boost domains such as business intelligence. Research into Natural Language Interfaces to Databases (NLIDB) has had a long history and can be tracked to the beginning of AI as a discipline [2]. Despite the long-standing research efforts, progress has been slow and widespread commercial use hasn't picked up.

The main issues plaguing the progress are due to language ambiguity and portability [3]. Another issue with data-driven approaches is that applying data-hungry models to the NLIDB problem is difficult due to the lack of large parallel corpora consisting of (text, SQL code) pairs. However, recent datasets alleviate this problem [4], [5].

The standard neural approach for code generation is a SEQ2SEQ architecture that maximizes the probability of the target code conditioned on the textual input [6].

Recent solutions incorporate syntactic information about the target code in order to improve the output. Instead of generating the code directly, [1] and [7] rely on the target language grammar to generate the underlying Abstract Syntax Tree (AST).

We modify the syntax-guided model of [1] so that it also handles schema information. The model outperforms a SEQ2SEQ baseline on the WIKISQL corpus, but not on the SENLIDB corpus. We show that this is due to the complexity of the latter dataset.

II. RELATED WORK

The NLIDB problem can be seen as an instance of semantic parsing, which is concerned with mapping a natural language utterance to a formal meaning representation. Semantic parsing has been traditionally deployed in question-answering systems, where the target is a logical form, but it has also been applied to code generation, where the output is a source code in a formal language such as PYTHON or SQL.

Neural approaches dominate the recent solutions in semantic parsing, mainly through the use of variants to the SEQ2SEQ architecture [8], [9]. This approach alleviates the need for building intermediate representations of meaning at the expense of requiring large parallel corpora of (annotation, logical form) pairs.

More advanced solutions [10] apply pointing mechanisms [11] to copy named entities from the textual description. Reinforcement learning is leveraged in [4] through a policy-based learning algorithm coupled with a pointing mechanism. They obtain diverse alternations to the *where* clause of an SQL query, while outperforming a strong neural machine translation baseline.

Recent approaches incorporate syntactic information about the target code in order to improve the output. Instead of generating the code directly, [1] and [7] rely on the target language grammar to generate the underlying Abstract Syntax Tree (AST). This ensures that the output is grammatically correct. Another approach in [12] is to decompose SQL generation problem into sketch generation and slot filling. An SQL sketch is obtained from the original SQL query by replacing specific columns and tables with placeholders. They show that decoding the SQL sketch first and then injecting it into a separate decoder improves the generation of the target SQL.

III. NEURAL MODEL

We follow the model in [1] and estimate the probability $P(y_{ast}|x, c)$, where y_{ast} is the underlying AST of the target SQL code y, x is the user question and c is the query-specific schema information. We factorize this as:

$$P(y_{ast}|x, c) = \prod_t P(a_t|a_{<t}, x, c),$$

where a_t are actions that construct the underlying AST of the code snippet y.

An action a_t can represent either the application of a production rule from the SQL grammar or the generation of

a token. A production rule consists of a left-hand side (non-terminal node) and a right-hand side containing non-terminal and/or terminal nodes. For instance, the production rule of a simplified SELECT statement can look like this:

$$select \rightarrow column\ from\ where$$

where *select*, *column*, *from* and *where* are non-terminal nodes.

We show in Fig. 1(a) the AST for the SQL code *select salary from employees where age > 5*. The sequence of actions corresponding to the AST generation is $[select \rightarrow column\ from\ where, column \rightarrow$ ***'salary'***$, $***'salary'***$ \rightarrow from, from \rightarrow table, ..., constant \rightarrow$ ***'5'***$]$.

The bolded nodes are terminals and correspond to string tokens. A token can be either generated from a fixed vocabulary or copied from the input using a copy mechanism [13], [14]. The model essentially learns the sequence of actions resulting from a depth-first traversal of the AST.

A. Input embedding

Let $X_{user} = [x^u_1, x^u_2, ..., x^u_U]$ be the user question tokens, $X_{columns} = [x^c_1, x^c_2, ..., x^c_C]$ be the column names, where $x^c_j = [x^c_{j,1}, x^c_{j,2}, ..., x^c_{j,T_j}]$ is the sequence of tokens in the j^{th} column. (e.g. "player no #" column results in the sequence ["player", "no", "#"]). Let L_j be the number of characters in the column name x^c_j.

Column representation We embed the j^{th} column by concatenating the average word and character embeddings:

$$emb(x^c_j) = [\frac{1}{T_j}\sum_{k=1}^{T_j} W_{word} \cdot x^c_{j,k}; \frac{1}{L_j}\sum_{i=1}^{L_j} W_{char} \cdot x^c_j[i]]$$

where W_{word} and W_{char} are pre-trained word and character embedding matrices.

We embed the user question tokens using a different matrix $W_{question}$.

B. Input representation

We encode the input differently for each dataset. For WIKISQL, the schema information (column names of Wikipedia tables) is specific to each example and so the input is $X = [X_{user}; X_{column}]$, where ; denotes the concatenation of the two sequences.

On the other hand, the SENLIDB queries correspond to a single database schema, so there is no need to encode the same columns for each example. We let $X = [X_{user}]$ be the input in this case.

To encode the input, we run a bidirectional LSTM network on tokens from X.

C. Decoder

The decoder is a modified LSTM that accommodates several inputs: context vector c_t, embedding of current node, embedding of current action a_t and previous hidden state s_{t-1}, embedding of parent action and parent hidden state.

D. Apply rule

The probability of selecting rule r is computed as:

$$P(a_t = ApplyRule[r]|a_{<t}, X) = softmax(W_{rule} \cdot s_t + b_{rule}) \cdot onehot(r)$$

where s_t is the current decoder state.

E. Token generation

Token prediction can be rewritten by marginalizing over the binary variable sel, which indicates whether token y_t is generated from the vocabulary or copied from the input:

$$P(a_t = GenToken[y_t]|a_{<t}, X) = P(sel = vocab|a_{<t}, X) \cdot P(y_t|vocab, a_{<t}, X) + P(sel = copy|a_{<t}, X) \cdot P(y_t|copy, a_{<t}, X)$$

The soft selection is modeled as:

$$P(sel|a_{<t}, X) = softmax(W_{sel} \cdot [c_t; s_t] + b_{sel})$$

where c_t is the context vector (encoder states weighted by the attention scores).

Vocabulary prediction is computed as:

$$P(y_t|vocab, a_{<t}, X) = softmax(W_{vocab} \cdot (W_{lin} \cdot [c_t; s_t] + b_{lin}) + b_{vocab})$$

The copying probability is computed similarly to the attention scores, by computing a soft- alignment between the encoder hidden states and the current hidden state s_t.

F. Training

We maximize the log-likelihood of the correct sequence of actions (*apply rule* or *generate token*) conditioned on the input (user request and table columns), over the entire training corpus D:

$$L(D) = \frac{1}{|D|}\sum_{i=1}^{|D|} \log P(y^{(i)}_{ast}|x^{(i)}, c^{(i)})$$

G. Inference

At inference we use the decoding algorithm of [1] to generate the surface form SQL from the underlying AST. Briefly, the algorithm is a modified beam-search that takes into account the type of the current node being expanded. If the current node is non-terminal then only production rules will be considered. Otherwise if the node is terminal, the network will generate a token from the fixed vocabulary or copy an input token. The algorithm terminates when all the ASTs in the beam are completed.

IV. EXPERIMENTS

We test our approach on two existing datasets, WIKISQL [4] and SENLIDB [5]. Our implementation and pretrained models are available at https://github.com/johnthebrave/code-generation.

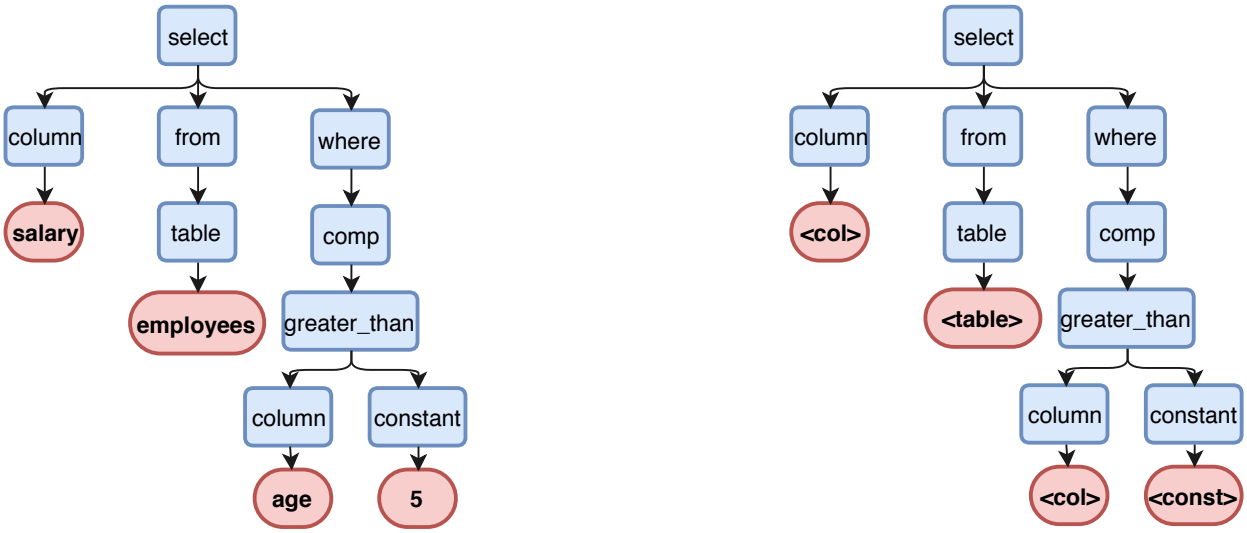

Fig. 1: (a) The AST for the SQL code *Select salary from posts where age > 5*. The blue rectangles correspond to non-terminal nodes, while the red ellipses correspond to terminal nodes. (b) The AST sketch corresponding to the previous SQL code. The values in the terminal nodes are replaced with placeholders. We can deterministically obtain a list-based logical form sketch from the AST sketch: *select <col> from <table> where <col> > <const>*

A. Datasets

The WIKISQL dataset features (question, SQL query) pairs over tables extracted from Wikipedia. The queries are fairly simple SELECT statements, with one or more WHERE clauses. Each WHERE clause follows a fixed pattern: *column condition constant*, where *condition* is one of the operators: $=$, $<$ or $>$. Moreover, each query is associated with a specific table, which is known beforehand. Therefore, the main difficulty lies in generating the appropriate columns and constants in the WHERE clause as well as the aggregation operator for the SELECT clause, if one is necessary.

The SENLIDB corpus contains (question, SQL query) pairs crawled from the website https://stackexchange.com. The queries are more difficult than in the WIKISQL corpus, featuring SQL clauses such as JOIN operations or nested SELECT statements. In Table I we list statistics for both datasets. The SENLIDB grammar has substantially more production rules (1154) than the WIKISQL grammar (55). The average number of actions is also larger (161.0 vs 59.6), with a larger spread around the mean (standard deviation 98.0 vs 13.3). On the other hand, the average input length on SENLIDB is smaller than WIKISQL (7.88 vs 11.7). Thus, the larger grammar size, longer sequences of target actions and shorter input descriptions of the SENLIDB corpus increase the burden on the decoder and make the generation process more difficult.

To process both datasets we used the same comprehensive grammar, designed for the T-SQL dialect. However, if we take into account the specific syntactical limitations imposed on the WikiSQL queries, then the number of production rules and implicitly the size of the ASTs may be significantly reduced.

STATISTIC	WIKISQL TRAIN	SENLIDB TRAIN
avg actions length	59.6	161.0
std actions length	13.3	98.0
avg AST size (#nodes)	51.78	135
std AST size	11.14	81
number of production rules	55	1154
avg input tokens	11.7	7.88

TABLE I: Statistics of the WIKISQL and SENLIDB datasets

B. Preprocessing

The SENLIDB queries have aliases, which are temporary names given to tables or columns. We identify such aliases, determine their type and index and replace them with a placeholder. For instance, the query SELECT *id as user_link* FROM *posts p* becomes SELECT *id as alias_col_1* FROM *posts alias_table_1*. The placeholders are then added to the target vocabulary.

C. Evaluation

We measure the logical form accuracy of the SQL queries on the WIKISQL and SENLIDB test sets. To obtain the SQL query, we pick the most probable AST using beam-search, and then deterministically convert the AST to the corresponding SQL.

We also report the sketch accuracy. We distinguish between two types of sketches: logical form sketch (list-based) and AST sketch (tree-based).

Logical form (LF) sketch Consider the following query: SELECT *student_gpa* FROM *students* WHERE *student_age* >

20. The corresponding LF sketch is: SELECT <*column*> FROM <*table*> WHERE <*column*> > <*constant*>. The LF sketch can be obtained in two ways: train a SEQ2SEQ network on (description, sketch) pairs directly and obtain the sketch [15] or train a SEQ2SEQ network on (description, code) and extract the sketch from the decoded code (as performed in the ONESTAGE model of [12]).

AST sketch There are also two ways to obtain an AST sketch. The first one is to remove the terminal nodes (nodes that hold table names, columns, string constants, etc.) from the original AST. Thus, the resulted *simplified* tree only captures the syntactic aspects of the query. We then compute the AST sketch accuracy by transforming the AST sketch into a sequence of actions and checking whether it's the same sequence that makes the ground truth AST sketch. This allows us to compare whether two queries have the same underlying structure, albeit with differently instantiated values. The second way to obtain an AST sketch is to directly generate it. Fig. 1(b) shows the AST sketch corresponding to the AST in Fig. 1(a). The values in the terminal nodes were replaced with appropriate placeholders: column names with <*col*>, table names with <*table*>, numeric and string constants with <*const*> etc. Thus, the size of the target vocabulary is significantly reduced, making it easier for the decoder to generate the AST sketch than the full AST. The AST sketch can be deterministically converted to an LF sketch, which is easier to compare to the ground truth LF sketch. For instance, the AST sketch in Fig 1(b) results in the LF sketch *select* <*col*> *from* <*table*> *where* <*col*> > <*const*>.

V. RESULTS

We report the logical form accuracy on both datasets in Table II. We compare the syntax-guided model to a neural machine translation (NMT) baseline reported by [15], a dual encoder sketch-based approach by [15] and to the coarse-to-fine (C2F) approach by [12].

On WIKISQL, our model increases the accuracy over the NMT baseline by 6.69% and is comparable to the sketch-based approach in [15]. This suggests that the benefits of simultaneously generating a syntax-and semantics-aware representation of code (AST) are similar to separating the syntactical (sketch generation) and semantic aspects (slot filling) of the generation process.

The current state-of-the-art approach in [12] significantly outperforms the syntax-guided method on the WIKISQL dataset. To achieve this result they introduce several model improvements (table-aware input encoder, fine meaning decoder).

However, since their solution is tailored to exploit the very specific nature of the SQL queries from WIKISQL, it cannot be applied on the much more complex SENLIDB corpus. For example, they formulate the sketch generation process as a classification problem. Therefore, they use a softmax classifier to choose from one of the 35 possible sketches of the WHERE clause, identified in the training set. For the SELECT clause, they use another softmax classifier to choose the aggregation operator and the associated column. This approach is unfeasible for queries where a more flexible grammar is required, as the number of possible sketches grows exponentially with the number of production rules.

METHOD	WIKISQL TEST	SENLIDB TEST
NMT [15]	32.07	2.44
DUAL ENCODER [15]	38.99	**3.97**
C2F [12]	**71.7**	-
Syntax-guided model	38.76	0.38

TABLE II: Accuracy of the generated SQL statements using the ground truth SQL as reference

METHOD	WIKISQL TEST	SENLIDB TEST
NMT [15]	82.38	**5.12**
C2F [12]	**95.9**	-
Syntax-guided AST sketch	-	1.69
Syntax-guided AST (–terminals)	86.74	1.03

TABLE III: Accuracy of the generated SQL sketches; the first three rows show LF sketch accuracies, while the fourth row shows an AST sketch accuracy, which is measured by looking at the action sequences instead of the SQL tokens

On SENLIDB, the syntax-guided model surprisingly fares worse than the NMT baseline. We suspect that the possible improvements brought by the syntactic guidance may be cancelled by semantic errors (i.e., incorrect tables or column names). We look at sketch accuracies in Table III to gain a better understanding of the model limitations.

Sketch accuracy The NMT baseline reported by [15] obtains 82.38% sketch accuracy on WIKISQL and only 5.12% on SENLIDB, which shows that it's more difficult to get the structure right for queries in the latter dataset.

On WIKISQL, the syntax-guided model obtains an AST sketch accuracy of 86.74%, higher than the 82.38% LF accuracy of the NMT baseline. This suggests that the NMT-based sketch generation step in [15] and [12] can be replaced by a syntax-guided approach in order to generate better sketches, which could also boost the query refinement step.

On SENLIDB, the accuracy of the AST sketches obtained from the full ASTs is very low (1.03%). To our surprise, the model trained to directly generate the AST sketch obtains 1.69% accuracy, which is also lower than the NMT baseline accuracy. Even though this model has a greatly reduced vocabulary output, it still cannot recover the underlying AST sketch, which is an easier problem than generating the correct AST (and corresponding SQL). We suspect that the larger grammar output (1154 production rules) and the larger AST size (135 nodes on average) for the SENLIDB prove too difficult for the decoder, which has to generalize to a much larger tree-output space.

VI. Conclusions

In this paper, we adapted a syntax-guided neural model to the NLIDB problem. The model takes into account schema information for the WikiSQL dataset, by encoding the column names together with the input description. Results on WikiSQL show that the syntax-guided model performs similarly to the sketch-based approach in [15]. As an interesting extension, we will investigate a merge between the two strands of work, by consecutively generating the AST sketch and then instantiating the terminal nodes. Additionally, we consider embedding type information, to help guide the inference process. That is, we would like to avoid a scenario where an operator or a function would be applied on a set of incompatible arguments.

The results on SenlIDB show that the syntax-guided model doesn't outperform the NMT baseline, due to the increased complexity of the dataset grammar relative to the dataset size. As future work here, unsupervised pre-training on a large amount of ASTs before conditioning on the user's description could boost performance of the SQL generation.

Acknowledgments

This work has been funded by the Text2NeuralQL research project (PN-III-P2-2.1-PTE-2016-0109).

References

[1] P. Yin and G. Neubig, "A Syntactic Neural Model for General-Purpose Code Generation," 2017. [Online]. Available: http://arxiv.org/abs/1704.01696

[2] I. Androutsopoulos, R. Graeme, and P. Thanisch, "Natural language interfaces to databases," *Proceedings of the twenty-second annual computer personnel research conference - CPR '86*, no. 709, pp. 12–26, 1986. [Online]. Available: http://dl.acm.org/citation.cfm?id=317210.317219

[3] R. Pazos, A. Rodolfo, B. González, J. Juan, A. L., A. Marco, F. Martínez, A. José, H. Fraire, and J. Héctor, "Natural language interfaces to databases: An analysis of the state of the art," *Studies in Computational Intelligence*, vol. 451, pp. 463–480, 2013.

[4] V. Zhong, P. Alto, C. Xiong, P. Alto, R. Socher, and P. Alto, "Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning," 2017.

[5] F. Brad, R. C. A. Iacob, I. Hosu, and T. Rebedea, "Dataset for a neural natural language interface for databases (NNLIDB)," in *Proceedings of the Eighth International Joint Conference on Natural Language Processing, IJCNLP 2017, Taipei, Taiwan, November 27 - December 1, 2017 - Volume 1: Long Papers*, 2017, pp. 906–914. [Online]. Available: https://aclanthology.info/papers/I17-1091/i17-1091

[6] L. Mou, R. Men, G. Li, L. Zhang, and Z. Jin, "On End-to-End Program Generation from User Intention by Deep Neural Networks," *Arxiv*, no. March 2016, 2015. [Online]. Available: http://arxiv.org/abs/1510.07211

[7] M. Rabinovich, M. Stern, and D. Klein, "Abstract Syntax Networks for Code Generation and Semantic Parsing," 2017. [Online]. Available: http://arxiv.org/abs/1704.07535

[8] R. Jia and P. Liang, "Data Recombination for Neural Semantic Parsing," pp. 12–22, 2016.

[9] L. Dong and M. Lapata, "Language to Logical Form with Neural Attention," 2016. [Online]. Available: http://arxiv.org/abs/1601.01280

[10] W. Ling, E. Grefenstette, K. Moritz Hermann, T. Kocisky, A. Senior, F. Wang, and P. Blunsom, "Latent Predictor Networks for Code Generation," *Acl*, pp. 1–13, 2016.

[11] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer Networks," pp. 1–9, 2015. [Online]. Available: http://arxiv.org/abs/1506.03134

[12] L. Dong and M. Lapata, "Coarse-to-Fine Decoding for Neural Semantic Parsing," 05 2018. [Online]. Available: https://arxiv.org/abs/1805.04793

[13] C. Gulcehre, S. Ahn, R. Nallapati, B. Zhou, and Y. Bengio, "Pointing the Unknown Words," *Acl 2016*, pp. 140–149, 2016. [Online]. Available: http://arxiv.org/abs/1603.08148

[14] J. Gu, Z. Lu, H. Li, and V. O. K. Li, "Incorporating Copying Mechanism in Sequence-to-Sequence Learning," *Acl*, p. 11, 2016. [Online]. Available: http://arxiv.org/abs/1603.06393

[15] I. Hosu, R. C. A. Iacob, F. Brad, and T. Rebedea, "Natural language interface for databases using a dual-encoder model."

Appendix