Text-to-SQL using Large Language Model
Project Report (Group 13)
CS839 Spring 2023

Harsh Sahu hsahu@wisc.edu
Sumedha Joshirao joshirao@wisc.edu
Anmol Sharma sharma265@wisc.edu
University of Wisconsin-Madison

Abstract

Natural language to SQL conversion is an important problem in the field of NLP and AI. Our project focuses on improving the accuracy of natural language to SQL conversion systems by fine-tuning a large language model. By providing the model with a natural language query and corresponding database schema, we aim to generate precise SQL queries. Our efforts will aid in bridging the gap between natural language and SQL and contribute to the advancement of NLP and AI.

In this project, we have fine-tuned T5 [10], a transformer-based large language model developed by Google, and pre-trained on a large set of varied text datasets including Wikipedia, news articles, books, etc, to work on a bunch of NLP tasks like Summarization, Question Answering, Text completion, etc. We evaluated our method on the SPIDER dataset [15], a benchmark dataset for Text-to-SQL tasks. Fine-tuning T5 on Spider, we were able to get significant performance improvement of at least 16%*, increasing from* 80% *to* 93%*. We, further, implemented rule-based approaches as a post-processing step to rectify common errors produced by the model, which further improved our scores to* 95%*.*

1. Introduction

Relational databases, particularly SQL databases, are widely used by organizations to store and manage their data due to their ease of use. SQL databases allow users to communicate with databases and perform various tasks like creating, updating, and deleting databases. This makes them a popular choice for any mid to large-sized company. However, retrieving data from SQL databases requires users to be skilled enough to write SQL queries. While this task may seem simple, it can be challenging as it often involves combining multiple tables and aggregating data to obtain the required information. This complexity makes writing SQL queries a difficult task. Additionally, many business stakeholders may not have the necessary skills or time to write these queries themselves. As a result, it would be highly beneficial if a system could convert natural language text to SQL queries automatically. This would make data retrieval from SQL databases easier and more accessible to a broader range of users, even those without advanced SQL skills.

NL: Show the names of students who have a grade higher than 5 and have at least 2 friends.

SQL:
```
SELECT T1.name
FROM friend AS T1 JOIN highschooler AS T2
ON T1.student_id = T2.id WHERE T2.grade > 5
GROUP BY T1.student_id HAVING count(*) >= 2
```

Figure 1. An example from the Spider benchmark to illustrate the mismatch between the intent expressed in NL and the implementation details in SQL. The column 'student id' to be grouped by in the SQL query is not mentioned in the question.

The task of Text2SQL involves translating a natural language question about a relational database into an SQL statement that can be used to query the database.

This is a significant challenge that can greatly enhance the accessibility of relational database management systems for end-users. By using Text2SQL techniques, we can create natural language interfaces for commercial RDBMSs, enabling easier access to database information. Ultimately, this can improve the user experience and make it easier for non-technical users to interact with complex relational databases.

The development of natural language interfaces for databases has been a subject of research in both the database and natural language communities for several decades. In the 1980s, methods were proposed that used intermediate logical representation to translate natural language queries into logical queries that could then be converted into database queries [12]. However, these methods still relied on manually crafted mapping rules for translation. In the early 2000s, more advanced rule-based methods were introduced. [9] utilized an off-the-shelf natural language parser to take advantage of advances in natural language processing without the need to train a parser for a specific database. However, this approach had limited coverage and was only able to handle semantically tractable questions.

In recent times, the NLP community has been actively researching deep-learning-based methods for natural language to SQL translation. These methods utilize advanced deep learning technologies. However, one of the primary challenges in developing a DL-based Text-to-SQL method is the scarcity of training data. There have been a lot of studies proposed to counter this problem. We discuss them in detail in 'Related Work'.

2. Large Language Model

With the recent developments in the field of Natural Language Processing (NLP), it was observed that the pre-trained Large Language Models when fine-tuned for a specific task, perform extremely well for that particular task. Also, the time and effort required to train and build a language model from scratch can be significantly reduced. One such LLM model is the Text-to-Text Transfer Transformer (T5) model. T5 is an encoder-decoder model that has been pre-trained on a combination of supervised and unsupervised tasks, which transforms any natural language processing problem into a text-to-text format. When provided with the appropriate prompts, T5 is a highly capable model that excels in a variety of tasks, including but not limited to text summarization, natural language inference, common sense reasoning, question answering, sentiment and sentence classification, translation, and pronoun resolution. Additionally, T5 is readily accessible without any cost, unlike models such as GPT-3. Based on these factors, we chose to fine-tune the T5 model for our Text-to-SQL conversion task.

Google released T5 in 5 different sizes. *t5-small, t5-base, t5-large, t5-3b, t5-11b*. Due to the limited GPU resources, we decided to use *t5-base* which has the least number of parameters.

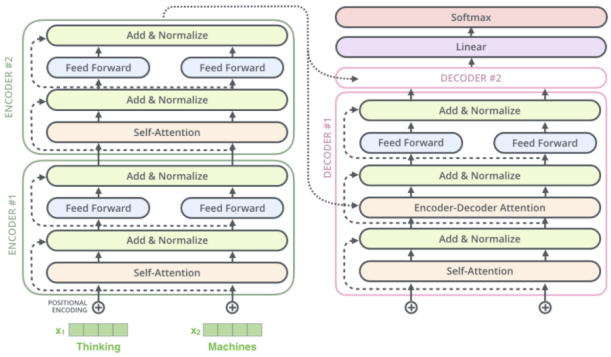

Figure 2. Encoder-Decoder architecture of T5

3. Dataset

We opted to use SPIDER dataset [15] for our project, which is currently the benchmark dataset for Text-to-SQL tasks. It is a large-scale, complex, and cross-domain text-to-SQL dataset that was introduced by researchers at Yale University in 2019. The Spider dataset consists of 10,181 natural language questions paired with 5,693 distinct SQL queries, spanning over 200 complex SQL schemas across 138 different domains. Figure 1 shows one of the examples of Spider Dataset. The SQL queries in the dataset can involve a wide range of SQL operations, including nested queries, aggregation functions, and joins. The training set contains 8,659 examples, the development set contains 1,034 examples, and the test set contains 488 examples.

As the test set is not publicly available, we train our

model on the training set and evaluated it against the development set.

The dataset is also categorized into four levels of difficulty, namely easy, medium, hard, and extra hard, based on the complexity of the SQL queries generated for each corresponding natural language query. The easy category includes queries with simple select or aggregation queries, while medium examples include a single join and an aggregation. Hard queries involve multiple join statements in the query, while extra hard queries require nested SQL queries in addition to multiple joins and aggregation functions. Figure 3 shows examples of queries at various levels of difficulty.

Easy

What is the number of cars with more than 4 cylinders?

```
SELECT COUNT(*)
FROM cars_data
WHERE cylinders > 4
```

Meidum

For each stadium, how many concerts are there?

```
SELECT T2.name, COUNT(*)
FROM concert AS T1 JOIN stadium AS T2
ON T1.stadium_id = T2.stadium_id
GROUP BY T1.stadium_id
```

Hard

Which countries in Europe have at least 3 car manufacturers?

```
SELECT T1.country_name
FROM countries AS T1 JOIN continents
AS T2 ON T1.continent = T2.cont_id
JOIN car_makers AS T3 ON
T1.country_id = T3.country
WHERE T2.continent = 'Europe'
GROUP BY T1.country_name
HAVING COUNT(*) >= 3
```

Extra Hard

What is the average life expectancy in the countries where English is not the official language?

```
SELECT AVG(life_expectancy)
FROM country
WHERE name NOT IN
   (SELECT T1.name
    FROM country AS T1 JOIN
    country_language AS T2
    ON T1.code = T2.country_code
    WHERE T2.language = "English"
      AND T2.is_official = "T")
```

Figure 3. Examples of Queries in Spider dataset basis different levels of difficulty

4. Evaluation Metric

It has been quite difficult for the researchers to comprehensively compare various studies in the domain of Text2SQL, because of the varied metrics that have been used. The evaluation strategies that have historically been used can be broadly classified into 4 categories: 1) string matching, 2) parse tree matching, 3) result matching, and 4) manual matching. String matching compares the ground truth or the GOLD query with the SQL query. When comparing a generated SQL query to the corresponding gold SQL query in string matching, certain factors can lead to incorrect judgments. For instance, the order of conditions in the WHERE clause, projected columns, and aliases can all play a role. Consider the example of two queries where one contains, $Q1$ AND $Q2$, in its WHERE clause and the other, contains $Q2$ AND $Q1$, where $Q1$ is "first_name = john" and $Q2$ is "age $\leq$ 32". In this case, even though the conditions are the same, the string match algorithm would still identify them as different queries due to the different order of the conditions.

Parse tree matching compares the parse trees of two queries. Although it has fewer errors than string matching, it can still be misleading. For instance, a nested query and its flattened version may be equivalent, but their parse trees would be distinct. Result matching is a method of comparing two SQL queries by executing them on the same database and comparing their results. This assumes that identical SQL queries will always produce the same results. However, there is a risk of overestimating the similarity of two different SQL queries if they happen to produce the same results by coincidence. Manual matching involves users verifying the accuracy of translation results by manually checking the execution results of SQL queries. However, this process requires a significant amount of manual effort and may not always guarantee reliability.

For our study, we use an improved version of string matching, *COMPONENT MATCHING* which is also one of the official metrics of Spider Dataset. It is the average exact match between the predicted and ground truth values for various SQL components. These components include SELECT, WHERE, GROUP BY, ORDER BY, and AND/OR (which is the aggregator of

conditions within the GROUP clause).

To evaluate each of these components, we break them down into sets of sub-components for both the predicted and ground truth values. For example, to evaluate a SELECT component such as "SELECT avg(col1), max(col2), min(col1)", we first parse and decompose it into a set of "(avg, min, col1)" and "(max, col2)", and then compare the sets from the predicted and ground truth values to determine if they match exactly.

Previous approaches have directly compared the decoded SQL with the ground truth SQL, but this method does not take into account SQL components that do not have order constraints. In our evaluation, we treat each component as a set so that variations in the order of the components do not affect the evaluation. For example, "SELECT avg(col1), min(col1), max(col2)" and "SELECT avg(col1), max(col2), min(col1)" would be treated as the same query.

To report the overall performance of our model on each component, we compute the precision score based on exact set matching.

5. Related Work

In the past few years, Text2SQL has become a popular research topic in both the database and natural language communities. Researchers from the database community such as [8] and [11] have proposed rule-based methods that utilize mappings between natural language words and SQL keywords. Meanwhile, [1] have improved the accuracy of these mappings and the inference of join conditions by using co-occurrence information of SQL fragments from query logs.

The evaluation of Text2SQL methods in a comprehensive manner has been a difficult task because different benchmarks have been used in various studies. For instance, while the WikiSQL benchmark has been used by [13] and [4], the GeoQuery benchmark has been used by [7]. On the other hand, the Spider benchmark has been employed by [5] and [3].

In recent years, the Text2SQL problem has gained attention from the natural language community, with several studies [5, 6, 13] using state-of-the-art deep learning models and algorithms. However, a major challenge for DL-based Text2SQL methods is the lack of training data. To address this, NSP [7] used an interactive learning algorithm and a data augmentation technique with templates and paraphrasing. DBPal [2] used a similar data augmentation technique with more varied templates and diverse paraphrasing methods. With the release of the WikiSQL benchmark, many studies have focused on improving accuracy on this dataset. For example, PT-MAML [6] used meta-learning, while IRNet [5] and SyntaxSQL-Net [14] were proposed for the SPIDER dataset.

6. Approach

In general, text-to-SQL conversion task can be divided into the following steps: 1) Fine-tune the selected LLM model on the training dataset for a selected number of epochs 2) Run predictions on the test data 3) Validate the results on some validation suite against the gold dataset and get the accuracy.

The following section explains our approach in detail. In order to tackle the problem, we decided to gradually build our approach in terms of the following steps:

1. Use the T5 model as it is without fine-tuning, get the predictions on the dataset, and measure performance

2. Fine-tune T5 model on the training data set without schema information and measure performance

3. Fine-tune T5 model on training data set along with corresponding database schema information and measure performance

4. Analyze model's output and markdown common errors across various queries. Devise and apply rules that correct these predictions as a post-processing step. Finally, evaluate the outputs.

Figure 4 shows the complete setup of our approach.

7. Using baseline T5

In the first step, we took *t5-base* and directly got the predictions on the Spider dataset. In this step, we did not train the model in any way or have given the database schema alongside the natural language query. As expected, this did not perform well as it was difficult for the model to generate the SQL queries without any training on the task

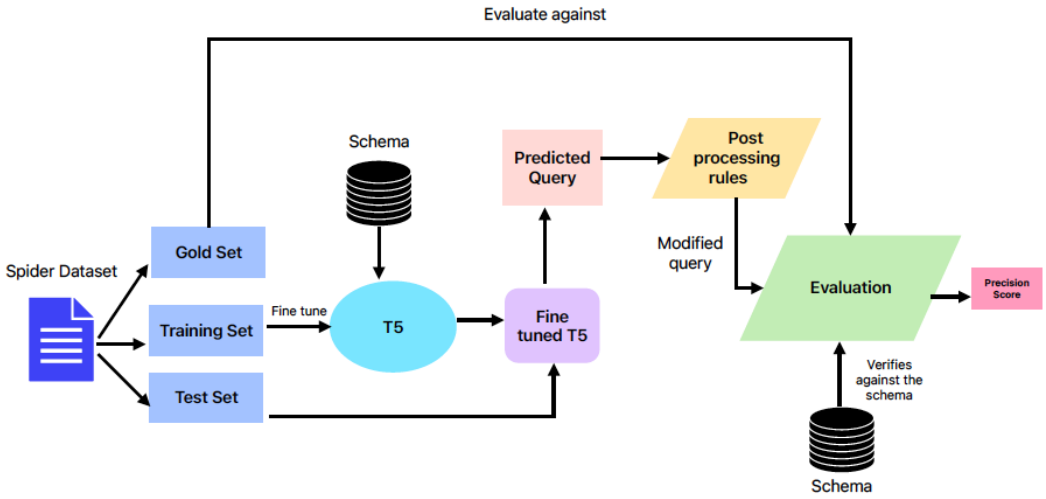

Figure 4. The complete Setup of our Approach

8. Fine-tuning T5 on NL query

8.1. Pre-Processing

Table 1 shows the information on the attributes that are being provided in the SPIDER dataset.

question	Question in natural language
question_toks	Natural Language question tokens
db_id	The Database id to which the Natural Language question is addressed
query	SQL query corresponding to the question
query_toks	SQL query tokens corresponding to the question
sql	Parsed results of this SQL query

Table 1. Attributes in SPIDER dataset

In the pre-processing step, we make use of the question, query, db_id column. To get the corresponding database schema for each query, we make use of the *tables.json* file included with the Spider dataset. Table 2 provides details about the database schema.

db id	database id
table names original	Original table names stored in the database
table names	Cleaned and normalized table names
column names original	Original column names stored in the database
column names	Cleaned and normalized column names
column types	Data type of each column
foreign keys	Foreign keys in the database
primary keys	Primary keys in the database

Table 2. Description of a Database schemas

8.1.1 Tokenization

The T5-base pre-trained tokenizer from Hugging Face is used to tokenize both the inputs and labels in our approach. During pre-training, the T5 tokenizer learns a fixed vocabulary of subwords, which are then encoded as integer IDs for input to the T5 model. The T5 model is an encoder-decoder architecture that employs the *SentencePiece* tokenizer.

8.1.2 Training

During training, we use input IDs and attention masks for input and only input IDs for the labels. After these two steps, the model is fine-tuned for 15 epochs with $8,659$ data points for training and $1,034$ data points for evaluation. As T5 model requires the task to be inputted along with the input text, we concatenate every input query along with the task description "translate English to SQL: " as a prefix.

During evaluation, it was observed that the model was not performing that well as it was not able to detect and place the schema information correctly in the SQL query. This led us to the third step.

9. Fine tuning T5 on NL query + DB schema

The third step was fine-tuning the model using the training dataset and passing the schema information explicitly during this process. For each data point in the Spider training dataset, we match db_id

Gold Queries	Predicted queries	Observed Error(s)
SELECT count(*) FROM AIRLINES WHERE Country = "USA"	SELECT count(*) FROM airlines WHERE country = "USA"</s>	Unwanted token </s> included at the end of query
SELECT T2.name , count(*) FROM Friend AS T1 JOIN Highschooler AS T2 ON T1.student_id = T2.id GROUP BY T1.student_id	SELECT T1.name, count(*) FROM High_Schoolers AS T	Wrong aliasing and wrong table name
SELECT cell_mobile_number FROM Students WHERE first_name = 'Timmothy' AND last_name = 'Ward'	SELECT phone_number FROM addresses WHERE first_name = "Tim	Wrong Column Name
select distinct year from cars_data where weight between 3000 and 4000;	SELECT YEAR FROM cars WHERE weight <unk> 4000 AND YEAR > 3000	<unk> inseated of '<' and wrong table name
SELECT document_id , document_name , document_description FROM Documents	SELECT document_ID, document_name, document_description FROM Document	Wrong table name
SELECT count(*) FROM CAR_MAKERS AS T1 JOIN COUNTRIES AS T2 ON T1.Country = T2.Countryld WHERE T2.CountryName = 'france';	SELECT count(*) FROM Car_Makers WHERE continent = "	Where clause wrongly predicted for join queries
SELECT COUNT(*) FROM CARS_DATA WHERE Accelerate > (SELECT Accelerate FROM CARS_DATA ORDER BY Horsepower DESC LIMIT 1);	SELECT count(*) FROM cars WHERE accelerate = "accrob	Wrong where cluase
SELECT Name FROM teacher ORDER BY Age ASC	SELECT T1.name FROM course AS T1 JOIN teacher AS T2 ON	Join query wrongly predicted
SELECT song_name FROM singer WHERE age > (SELECT avg(age) FROM singer)	SELECT name FROM singer WHERE age > (SELECT avg(	Truncated query

Table 3. Common Errors in Predictions

with the corresponding db_id in *tables.json*. Then we join the column names original for each training data point. Given a natural language question Q_{nl}, and its corresponding database $D_q = (T, C)$ where $T = \{t_1, t_2, ..., t_n\}$ is the list of tables and $C = \{ct_1, ct_2, ..., ct_n\}$ is the list of columns in the corresponding database, where n refers to the total number of tables, and ct_k refers to the list of column names of k^{th} table. So, the final input that goes into the model looks something like:

$$\text{input} = \text{task-prefix} + Q_{nl} + \langle schema \rangle$$
$$+ t_1 + t_2 + \cdots + t_n + ct_1 + \cdots + ct_n$$

Here, task-prefix refers to "translate English to SQL: ". And, $\langle schema \rangle$ is a "special token" for the schema. So, everything including the natural language query, table names, and columns list is fed as input during training.

10. Prediction Analysis

We analyzed the predictions generated by the fine-tuned model and found some repeating common errors that the model was making: 1) The table names in the predicted sql queries as compared to the gold sql queries were incorrect 2) The column names in some queries were also incorrect 3) Some queries did not have proper aliasing 4) Unwanted tokens like <unk> were inserted in queries that were expected to have characters like '<' or '>' in the where clause. 5) Some correctly predicted queries included <pad> or </s> token at the beginning or the end of the query. 6) Model could not predict long queries, truncating the predictions after a certain number of characters 7) WHERE clause was wrongly predicted for complex join queries. 8) WHERE clause conditions were incorrectly predicted 9) JOIN clause was wrongly predicted for some queries.

Table 3 shows some of the instances where the above errors are made in the predictions by the model.

We fix some of these errors in the post-processing step.

11. Post-processing

11.1. Column/Table name correction

Going over the model's predictions, we noticed some repeating mistakes that the model was making in its outputs. One of them is the misspelling of column or table names. If we check Example 2 in Table 3, the model mistakes the table name *Highschooler* as *High_schoolers*. The same error occurs in example 3 mistaking *cell_mobile_number* as *phone_number*. Below is the rule-based logic that we built to correct column names:

1. Knowing the position of a column name being appeared in a query, we extracted all column names from the query. For example, all the terms men-

tioned after SELECT and before FROM. We used *RegEx* for this task.

2. We, then, fetched all the column names from the corresponding database schema

3. For each column name found in the query, we find its closest match in the columns extracted from the schema. We replace that column name with the match found.

We implemented the same logic for correcting the table names

11.2. Table aliasing correction

One of the other commonly found errors is the wrong alias or reference of a table used in different parts of the query. If we look at example 2 in Table 3, the model, first, assigns Table *Highschooler* as *T*, but then it mistakenly addresses its column *name* as *T1.name*. We built the following logic in order to fix such errors.

1. We built a dictionary of all the tables mentioned in the body of the predicted query and their assigned aliases

2. We get all the columns in the query, and find their corresponding source tables from the database schema. For columns present in multiple tables, we prefer the table for which it is the primary key

3. For each column, we then changed the table alias attached as the prefix, to the correct alias of the source table

11.3. Replacing <unk> with >,<

We also noticed that instead of outputting the supposed '<' or '>' sign in the WHERE clause, the model predicted '<unk>' in a lot of queries. One such case is shown in example 4 of Table 4. In order to correct this error, we needed to know the correct sign (either '<' or '>') to replace <unk> with. We rectified such error in the following way:

1. We build two vocabularies of all commonly used comparative words. The first vocabulary contains words that correspond to '<' like, less, smaller, etc. The second vocabulary contains words that correspond to '<' like, more, greater, etc.

2. Now, for each query with <unk> in WHERE clause, we identify the column on the left side of the <unk> token. We match the closest word for this column name in the original NL query

3. For the match, we then find one of the comparative words in its vicinity (2 words preceding and 2 words following). After finding the comparative word, we replace the <unk> token with the corresponding comparative sign.

4. As we have found much more instances of '<' being replaced by <unk> compared to '>'. In case we don't find a match, we simply replace <unk> with '<'.

11.4. Removing T5's tags

As can be seen in Example 1 of Table 3, T5 also outputs a lot of tags, like '$\langle\backslash s\rangle$', at the beginning and/or at the end of the output sentence. We pruned all such tokens from the beginning as well as the end of the predicted query by the model.

Table 4 shows the corrected predictions after the implementation of the post-processing steps.

12. Results

We measured the precision scores of our model using component matching and evaluated its performance at every step of our approach. To accomplish this, we used the spider validation test suite [16]. Figure 5 shows the comparison between the precision scores for each component SELECT, WHERE, GROUP BY, ORDER BY, and AND/OR with respect to the four approaches. We find that the performance of the T5-Base model without fine-tuning was very poor for all components except for AND/OR components. Baseline T5 simply fails to predict almost all the queries. This is expected as T5 is not pre-trained on such a task, and would have never seen a SQL query in its training. Being able to produce AND/OR clause may be attributed to the fact that T5 is pre-trained on tasks related to understanding English text, which allows it to capture AND/OR clauses accurately from the natural language query.

After performing pre-processing techniques and tokenization prior to fine-tuning the T5 model without explicitly including schema information, we noticed a considerable enhancement in scores for almost all the

Gold Queries	Predicted queries	Observed Error(s)
SELECT count(*) FROM AIRLINES WHERE Country = "USA"	SELECT count(*) FROM airlines WHERE country = "USA"	[Corrected]
SELECT T2.name , count(*) FROM Friend AS T1 JOIN Highschooler AS T2 ON T1.student_id = T2.id GROUP BY T1.student_id	SELECT T.name, count(*) FROM Highschooler AS T	[Corrected]
SELECT cell_mobile_number FROM Students WHERE first_name = 'Timmothy' AND last_name = 'Ward'	SELECT cell_mobile_number FROM addresses WHERE first_name = "Tim	[Corrected]
select distinct year from cars_data where weight between 3000 and 4000;	SELECT YEAR FROM cars WHERE weight < 4000 AND YEAR > 3000	[Corrected]
SELECT document_id , document_name , document_description FROM Documents	SELECT document_ID, document_name, document_description FROM Documents	[Corrected]
SELECT count(*) FROM CAR_MAKERS AS T1 JOIN COUNTRIES AS T2 ON T1.Country = T2.CountryId WHERE T2.CountryName = 'france';	SELECT count(*) FROM Car_Makers WHERE continent = "	Where clause wrongly predicted for join queries
SELECT COUNT(*) FROM CARS_DATA WHERE Accelerate > (SELECT Accelerate FROM CARS_DATA ORDER BY Horsepower DESC LIMIT 1);	SELECT count(*) FROM cars WHERE accelerate = "accrob	Wrong where cluase
SELECT Name FROM teacher ORDER BY Age ASC	SELECT T1.name FROM course AS T1 JOIN teacher AS T2 ON	Join query wrongly predicted
SELECT song_name FROM singer WHERE age > (SELECT avg(age) FROM singer)	SELECT name FROM singer WHERE age > (SELECT avg(	Truncated query

Table 4. Predictions after post-processing

components, except for the WHERE and ORDER BY components. Precision scores improved, ranging from 45% to 89%. However, after modifying the prompt input during fine-tuning through schema concatenation, the model's performance significantly improved, ranging between 80% to 93% for different components. Nonetheless, it was observed that the precision score for the WHERE component remained at 0%. This shows that the WHERE clause is the most difficult of all.

The blue bars in Figure 5 represent the precision scores obtained after applying post-processing rules. The scores improved for almost all components, including the WHERE component, though a slight decrease in precision was observed for the GROUP BY and ORDER BY components. The precision score comparison for the different levels, i.e., easy, medium, and hard queries from the test suite in the spider dataset after the post-processing step is presented in Figure 6. As per the figure, the model performed exceptionally well on all components except for the GROUP BY component in the easy query category. The model also performed well on the medium queries for all components. For hard queries, it performed well for all components except the order by component.

Unintuitively, the model worked gradually better on the GROUP BY clause as the difficulty of the query rose. We observe that for shorter queries, the model truncates the predictions without printing the WHERE clause, whereas for longer inputs (NL query), the

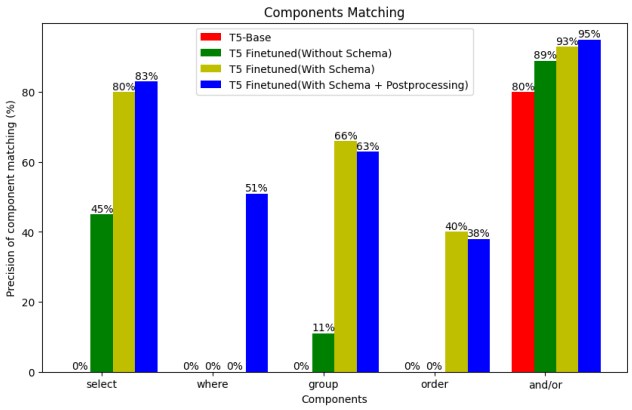

Figure 5. COMPONENT MATCHING score for each SQL component across various methods

model tends to produce longer SQL queries that would contain the WHERE clause. We suspect that this might be the reason for getting these unexpected results.

Overall, we see a gradual improvement in the results over our successive steps, which shows even more room for further improvement. We also see relative performances over various clauses as expected, as some are easy while others are hard to predict.

13. Challenges

The biggest challenge we faced is the training of the T5. Given the size of its architecture, we could not train it on our local systems, neither we could leverage freely available GPU resources like *Google Colab*

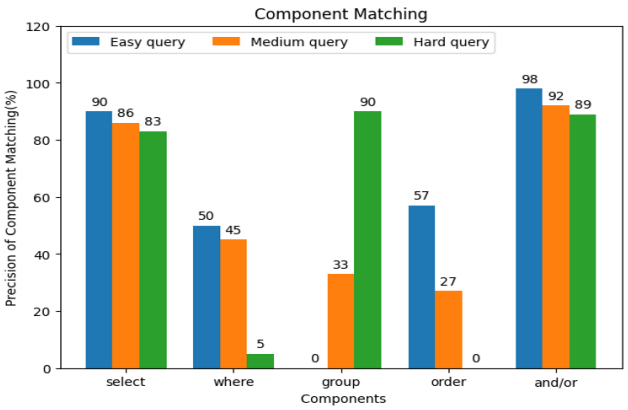

Figure 6. COMPONENT MATCHING score for each SQL component across different levels of query, using *T5 fine-tuned with schema + post-processing*

as the model was too big and the session kept crashing. We, finally, used Google Cloud Platform (GCP) to train the model.

We did inferencing and post-processing on freely available Google Colab notebooks.

14. Conclusion and Future Work

14.1. Conclusion

In this project, we fine-tuned the T5-base model to generate SQL queries from natural language text. Throughout the training process, we have demonstrated the evolution of our model's performance. At the beginning, when the T5 model was trained on the training set without any changes, it showed poor results across all the individual components, such as SELECT, WHERE, GROUP BY, and ORDER BY. Simply by adding some pre-processing steps and tokenization before training, the performance of the model jumped up for a few of the components which was quite impressive, given that the T5 model has not been pre-trained for this particular task. By modifying the prompt input during fine-tuning with the concatenation of schema information, we were able to improve the model's performance significantly from 80% to 93% for different components. Moreover, by applying minor corrections to the predicted queries, we were able to further increase the model's performance to 95% on the evaluation metric. The model was also now able to predict the where clause correctly 51% of the times which is a steep increase as compared to prior post-processing where the initial precision was 0% for WHERE clause.

14.2. Future work

As part of the prediction analysis, it was noted that certain queries produced by the model were either truncated or structurally incorrect. Therefore, the next potential advancement in this area could involve ensuring that the model generates queries that are not only complete but also syntactically accurate, and capable of being compiled without encountering errors.

References

[1] Christopher Baik, Hosagrahar V Jagadish, and Yunyao Li. Bridging the semantic gap with sql query logs in natural language interfaces to databases. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 374–385. IEEE, 2019. 4

[2] Fuat Basik, Benjamin Hättasch, Amir Ilkhechi, Arif Usta, Shekar Ramaswamy, Prasetya Utama, Nathaniel Weir, Carsten Binnig, and Ugur Cetintemel. Dbpal: A learned nl-interface for databases. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1765–1768, 2018. 4

[3] Ben Bogin, Matt Gardner, and Jonathan Berant. Representing schema structure with graph neural networks for text-to-sql parsing. *arXiv preprint arXiv:1905.06241*, 2019. 4

[4] Li Dong and Mirella Lapata. Coarse-to-fine decoding for neural semantic parsing. *arXiv preprint arXiv:1805.04793*, 2018. 4

[5] Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. Towards complex text-to-sql in cross-domain database with intermediate representation. *arXiv preprint arXiv:1905.08205*, 2019. 4

[6] Po-Sen Huang, Chenglong Wang, Rishabh Singh, Wen-tau Yih, and Xiaodong He. Natural language to structured query generation via meta-learning. *arXiv preprint arXiv:1803.02400*, 2018. 4

[7] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. Learning a neural semantic parser from user feedback. *arXiv preprint arXiv:1704.08760*, 2017. 4

[8] Fei Li and Hosagrahar V Jagadish. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment*, 8(1):73–84, 2014. 4

[9] Ana-Maria Popescu, Alex Armanasu, Oren Etzioni, David Ko, and Alexander Yates. Modern natural language interfaces to databases: Composing statistical

parsing with semantic tractability. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*, pages 141–147, 2004. 2

[10] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020. 1

[11] Diptikalyan Saha, Avrilia Floratou, Karthik Sankaranarayanan, Umar Farooq Minhas, Ashish R Mittal, and Fatma Özcan. Athena: an ontology-driven system for natural language querying over relational data stores. *Proceedings of the VLDB Endowment*, 9(12):1209–1220, 2016. 4

[12] David HD Warren and Fernando CN Pereira. An efficient easily adaptable system for interpreting natural language queries. *American journal of computational linguistics*, 8(3-4):110–122, 1982. 2

[13] Tao Yu, Zifan Li, Zilin Zhang, Rui Zhang, and Dragomir Radev. Typesql: Knowledge-based type-aware neural text-to-sql generation. *arXiv preprint arXiv:1804.09769*, 2018. 4

[14] Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir Radev. Syntaxsqlnet: Syntax tree networks for complex and cross-domaintext-to-sql task. *arXiv preprint arXiv:1810.05237*, 2018. 4

[15] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task, 2019. 1, 2

[16] Ruiqi Zhong, Tao Yu, and Dan Klein. Semantic evaluation for text-to-sql with distilled test suite. In *The 2020 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2020. 7