Natural Language Query to MongoDB Query

Bhavin Oza
Rochester Institute of Technology
Rochester, NY - 14623
bo2115@rit.edu

1 INTRODUCTION

Natural Language Interfaces is an evolving research area, aimed at learning and contextualizing the natural language processing for human computer interaction systems. With the advances in natural language processing (NLP) with machine learning, many significant systems have been built to understand and process human language and provide the necessary output in terms of code or database queries. Few of these systems which are remarkable are based upon works of the Transformer and its attention mechanism. Our project works on one such system, where we convert natural language queries to MongoDB queries.

1.1 Seq2SQL

In the Seq2SQL model[6], the approach adopted is that of sequence-to-sequence architecture with reinforcement learning. In order to learn a policy to construct a query that has unordered sections that are less suited for optimization via cross entropy loss, the model rewards from in-the-loop query execution across the database. The model is trained on a large set of natural language questions with SQL schema and queries, with the techniques combining supervised and reinforcement learning, where the reinforcement learning part plays a significant role in improving the accuracy of the SQL query generation. This sets a new standard demonstrating the effectiveness of the deep learning networks with the reinforcement learning technique, contributing to the evolving field of natural language processing.

1.2 Transformer

A revolutionary architecture for deep learning was introduced for the sequence to sequence tasks in the 'Attention is All You Need' paper by Vaswani et al, addressing and improving on the challenges in the traditional models like RNNs and CNNs for sequential processing.[5]. This architecture works on the self attention mechanism, consisting of encoder and decoder models for processing input and output having multiple layers of feed-forward neural networks and multi-head attention. The self attention mechanism defines the principle of allowing the model to consider the different input tokens from the tokenized string, which helps in understanding the dependencies between different tokens irrespective of their position in the input. The feed-forward networks are part of each layer, contributing to the non-linearity in the model, whereas the multi-head attention helps in parallelizing the need to attend to different combinations of representations. Thus this leads in creating complex natural language processing models, which provide more accuracy and efficiency.

Based on the attention mechanism defined above, transformer models were created. These models define natural language processing as a text to text problem, thus allowing a consistent approach for training across all such tasks. The initial model here Transformer T5[3] is trained on a large text corpus on unsupervised learning objectives. Then this model is leveraged for transfer learning. NLP has seen the rise of transfer learning, which is the process of pre-training a model on a task with plenty of data and then fine-tuning it on a particular job. This leads to less training time and more impressive results, for many tasks of text classification, language modeling, .etc.

Thus, building on the foundations and techniques mentioned above from different research projects, our project aims to develop neural network based NLQ system for MongoDB queries, creating an analytical tool for users with little to no background in MongoDB.

2 DATASET

2.1 Introduction

For training the model, we have considered an existing dataset WikiSQL[4] which has NLQ to SQL queries stored in a large JSON file. For the purpose of the simplicity of the training and evaluation, we have considered the simple SQL queries with WHERE clause and Accumulators with aggregate functions eliminating complex queries with join statements. Thus, the final dataset with NLQ and MongoDB queries has 56,050 records.

WikiSQL is a massive dataset that includes over 80,000 hand-annotated examples of natural language questions along with their corresponding SQL queries. This dataset has been meticulously curated, spanning over 24,000 tables derived from Wikipedia articles. Each question in the dataset is annotated with a SQL query, making it an invaluable resource for training and evaluating natural language interfaces for relational databases. The WikiSQL dataset make it a valuable resource for researchers and developers working on NLP tasks in database querying, allowing for progress in the development of more accurate and robust systems for translating NLQ into SQL.

Since, the original data is in SQL format, to meet the requirements of the dataset for our project, we have developed an algorithm for converting these SQL queries and schemas into their equivalent MongoDB queries and schema with appropriate data types. The algorithm focuses on cleaning, preparing, and initializing a dataset for NLQ to MongoDB queries conversion, which is defined as a JSON object.

The final structure of a record in the dataset used for training which is stored as JSON string is as follows:

```
{
  "instruction": {
    "schema": {
      "collections": {
        "collection_name": {
```

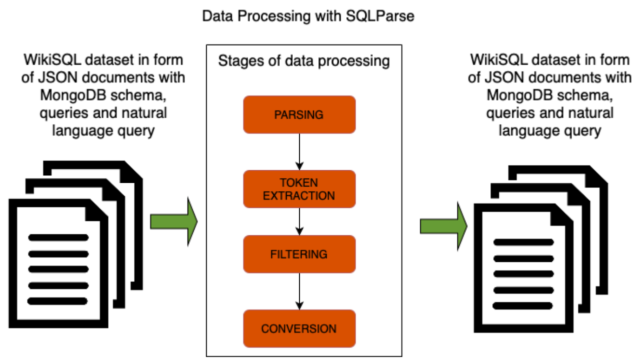
Data Processing with SQLParse
WikiSQL dataset in form of JSON documents with MongoDB schema, queries and natural language query
Stages of data processing
PARSING
TOKEN EXTRACTION
FILTERING
CONVERSION
WikiSQL dataset in form of JSON documents with MongoDB schema, queries and natural language query

Figure 1: Steps in Data Processing.

```
          "type": "object",
          "properties": {
            "field_name": {
              "type": "field_type"
            }
          }
        }
      }
    },
    "question": "natural language query"
  },
  "query": "MongoDB query"
}
```

2.2 Reading and Filtering

Reading the WikiSQL large JSON file containing the SQL components and questions, we iterate over each line in the file which is a JSON object, and parse each of these JSON objects and extract relevant information, which are SQL schema, NLQ, and SQL query output for further processing.

2.3 Conversion Of Schema

We parse the SQL Schema using the SQLParse[1] python framework, which represents each of the parts of the schema as a token. On parsing the SQL schema, we extract the table names as Identifier tokens and the columns in the Parenthesis token. We further process the Parenthesis token for the column names and its respective data types which are present as the sub tokens. For each of the SQL data types of columns we get its respective MongoDB data type, which is used for constructing the MongoDB schema. The MongoDB schema is in the form of JSON, where we have the 'collections' field and its value is a JSON of collection names and its fields stored in the 'properties' field, similar to the JSON schema we have for a MongoDB collection.

Given the following SQL schema:

```
CREATE TABLE teams (
    "School" text,
    "Team_Name" text,
    “Town" text,
    "County" text,
    "School_Enrollment" real,
    "Football" text
)
```

we will get the following list of tokens on parsing it using sqlparse:

(1) <token DDL 'CREATE'>
(2) <token Whitespace ' '>
(3) <token Keyword 'TABLE'>
(4) <token Whitespace ' '>
(5) <token Identifier 'school...'>
(6) <token Whitespace ' '>
(7) <token Parenthesis '(...)'>

The equivalent MongoDB collection schema we have from our algorithm is as follows:

```
"teams": {
  "type": "object",
  "properties": {
    "School": {
      "type": "string"
    },
    "TeamName": {
      "type": "string"
    },
    "Town": {
      "type": "string"
    },
    "County": {
      "type": "string"
    },
    "School_Enrollment": {
      "type": "double"
    },
    "Football": {
      "type": "string"
    }
  }
}
```

We remove all the whitespace tokens. The identifier with table name is stored in the collection schema for the MongoDB, and we further process the sub tokens from the Parenthesis token, which has the column name and types as the Identifier token. Due to the nature of some of the queries, at times the column name and types are stored in a single token as IdentifierList, we split these tokens, and then iterate through each of the tokens in the list to get column name and its data type. The column name is processed to replace any spaces with underscore, to make it easy for understanding and processing. The column data type is then converted to its respective MongoDB data type. These column names and data types are then stored in the form of field and type in the MongoDB collection schema. The final MongoDB schema is then stored in a dictionary with collection name as key and its properties and fields under its value. This gives a JSON structure which we will further dump as a JSON string in the final dataset file. The final schema will be subfield of the instruction field.

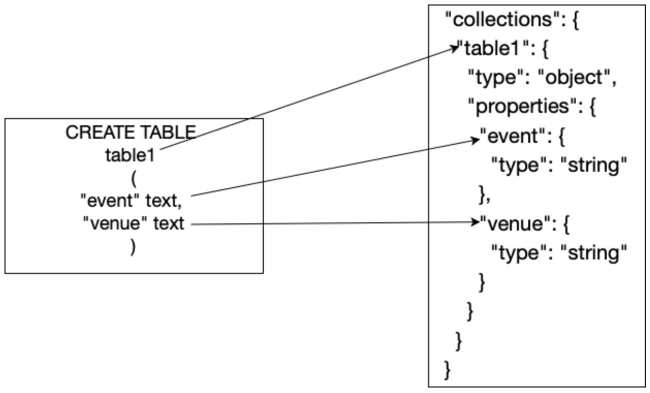

Figure 2: Schema Conversion SQL to MongoDB.

2.4 Conversion Of Query

For each SQL query, tokenize the query using SQLParse[1] framework in python. Based on the tokens present such as join, aggregate columns, we filter out and extract relevant tokens from the parsed query. Then we convert the filtered tokens into MongoDB-compatible queries using a set of rules, by considering only the selected column, where and comparison tokens from the query. If the conversion is successful, construct MongoDB-compatible queries.
For example, given the following query:

```
SELECT "School" FROM teams WHERE "TeamName" = "trojans"
```

we will get the following list of tokens on parsing it using sqlparse:

(1) <token SELECT>
(2) <token Whitespace>
(3) <token value = “School”>
(4) <token Whitespace>
(5) <token FROM>
(6) <token Identifier value = “teams”>
(7) <token WHERE, value = (WHERE "TeamName" = "trojans")>

On further dividing the WHERE token, we will get the column name and values as well, for example: "TeamName" and "trojans" as Identifier tokens.

We iterate through each of them and remove all the whitespace tokens, and we further process the sub tokens from the WHERE clause token, which have the Comparison token, along with the column as identifier and column name as Identifier sub tokens. We remove the white spaces from the sub tokens as well. The equivalent MongoDB query we have is as follows:

```
db.teams.aggregate(
[{
    $match:
        { "Team_Name" : "trojans"}
},
{
    $project:
        { "School" : 1}
}
])
```

2.5 Data Initialization and Output

In the output JSON file, we store the clean and transformed dataset which is used for training the models. These dataset has instructions with MongoDB schema and NLQ with its corresponding MongoDB queries. Each line in the output file represents a JSON object containing the instruction and MongoDB query.

3 METHODOLOGY

3.1 Model Architecture And Tokenization

The choice of model architecture has a significant impact on the success of a NLP task. The T5 (Text-To-Text Transfer Transformer) architecture is used here, which is a Transformer model variant. This architecture, which is well-known for its versatility and effectiveness across a wide range of NLP tasks, is implemented using the Hugging Face Transformers library. The model is loaded with pre-trained weights from the "juierror/flan-t5-text2sql-with-schema-v2" checkpoint[2], which has been fine-tuned for the text-to-SQL task. Tokenizers are used to ensure effective communication between the model and the dataset. This tokenizer, also provided by the Hugging Face Transformers library, converts raw text inputs into numerical representations that the model can understand. Custom tokens and are added to the tokenizer to address specific syntax requirements in MongoDB queries. Additionally, the tokenizer handles padding and truncation, ensuring that input sequences adhere to a maximum length of 512 tokens allowed by the model.

3.2 Training

In the pursuit of computational prowess, training is carried out seamlessly, utilizing the computational prowess of CUDA-enabled GPUs when available and gracefully degrading to CPU in their absence. A group of hyperparameters—batch size, learning rate, and epochs; are carefully selected to strike a delicate balance between model convergence and computational efficiency. The values for each of the hyperparameters is as follows: Batch Size(1), Learning Rate(0.0001) and Epochs(2). Each epoch represents a new opportunity for model refinement, as the entire training dataset is meticulously traversed. The AdamW optimizer, one of the famous optimization algorithms, oversees the model parameter optimization. At each iteration, the model is delicately fine-tuned, and the loss is computed using the discerning negative log-likelihood filter.

The training approach begins with importing data from the given large JSON file comprising of NLQ to MongoDB queries in the instructions, followed by partitioning the dataset into training and testing subsets with the train_test_split function. The separation of train and test allows for the evaluation of model performance utilizing previously unknown data during testing. The test size, which is set to 0.2, indicates that 20% of the data is maintained for testing and the other 80% for training. The random_state argument is set to 42 for consistency by reproducing the same sequence of random numbers each time we run the code. It initializes the random number generator used for data splitting.

Subsequently, the training and testing datasets are processed into instances of a custom dataset tailored for the specific task at hand. The next step involves tokenizing the input data using a provided tokenizer, which is known as tokenization. It is a preprocessing step in NLP, where text data is converted into numerical tokens suitable for consumption by machine learning models. The DataLoader class is used to create iterable batches of data for both training and testing. Thus, this leads to efficient data loading and processing, especially when dealing with large datasets that may not fit entirely into memory.

Within the training loop, the model is put into training mode, which ensures that layers like dropout and batch normalization behave differently during training compared to evaluation. The training loop iterates over multiple epochs, with each epoch comprising iterations through batches of data. For each batch, the input sequences, attention masks, and target sequences are extracted and transferred to the specified computing device (e.g. GPU) to leverage hardware acceleration if available.

During the optimization phase, the model's forward pass is carried out to generate predictions, and the optimizer's gradients are reset. The differences between the model's output and the real data are captured by calculating the loss by comparing the model's predictions with the target sequences. The model's performance on the training set of data is shown by this loss value.

Next, backpropagation is used to compute the gradients of the loss function with respect to the model's parameters. These gradients are used by the optimizer to modify the model's parameters in a way that minimizes loss and improves performance. Gradient descent is a crucial training technique for deep learning models, as it allows for constant parameter adjustments.

3.3 Accuracy Calculation

We use a custom approach to calculate accuracy that differs from direct string comparison. This is necessary, because there may be differences in whitespaces between actual and predicted query strings, resulting in inconsistencies despite the fact that both are correct. Our approach involves converting actual and predicted queries into collection names and aggregate pipelines of MongoDB involving the query stages. Subsequently, we compare these components recursively, examining each stage in pipeline within the list of pipelines represented as JSON dictionaries. We determine query correctness by recursively comparing keys and values, which includes nested dictionaries and primitive values. Any discrepancy marks the predicted query as incorrect, whereas identical comparisons mark it correct. This meticulous process ensures accurate counting of predicted queries, allowing for an accurate assessment of testing batch accuracy.

4 EVALUATION

We achieved an accuracy rate of 75.18% throughout the model testing phase, indicating excellent performance. This implies that the model understands and translates user queries quite well, producing MongoDB queries with a high degree of success. Considering the variety of query types present in the dataset, we concentrated on those that were relevant to MongoDB, specifically WHERE clause-based searches and queries that involved Acummulators. The distribution of all records and the corresponding accuracy for each query class are shown in the table below.

Table 1: Accuracy

	Simple query	Accumulator query
Accuracy Percentage	81%	59%
Total Test Records	8660	2550

Below is an example of the correct prediction of MongoDB query by model when given a JSON string representing instruction with schema and question:

```
Given Instruction as JSON String:
{
  "instruction": {
    "schema": {
      "collections": {
        "table1": {
          "type": "object",
          "properties": {
            "event": {
              "type": "string"
            },
            "venue": {
              "type": "string"
            }
          }
        }
      }
    },
    "question": "What is the 2022 Fifa Final venue?"
  }
}

Correctly Predicted Query:
db.table1.aggregate([
  {
    "$match": {
      "event": "2022 Fifa Final"
    }
  },
  {
    "$project": {
      "venue": 1
    }
  }
])
```

In spite the model's ability to generate accurate queries based on its training, several instances of incorrect queries were produced. These inaccuracies were frequently caused by erroneous queries with incorrect collection names or field value capitalization. For example, the expected query

```
db.table_2661.aggregate(
[{
    "$match": {
        "date": "2021 March"
    }
}]
)
```

was incorrectly generated as

```
db.table_2641.aggregate(
[{
    "$match": {
        "date": "2021 march"
    }
}]
)
```

Furthermore, discrepancies were discovered during the $match stage, with produced queries containing different field values than anticipated. For instance, an expected $match stage of

```
{
    "$match": {
        "Segment_D": "Pedal Steel Guitars"
    }
}
```

was actually produced as

```
{
    "$match": {
        "Episode": "111"
    }
}
```

Although several aggregate function queries returned the expected results, some were inaccurate due to incorrect accumulators. For example, an expected query of

```
{
    "$group": {
        "_id": "null",
        "avg": {
            "$avg": "$2012"
        }
    }
}
```

produced an incorrect accumulator as

```
{
    "$group": {
        "_id": "null",
        "max": {
            "$max": "$2012"
        }
    }
}
```

5 FINAL REMARKS

We've developed a deep neural network that translates questions into MongoDB queries, using transfer learning on a pre-existing Transformer model. Our approach builds on MongoDB query structure to grasp its context within NLQ. Further enhancements in model accuracy can be achieved through diverse set of query examples and using larger language models with more parameters, making it a valuable analytical tool for non-technical MongoDB users.

REFERENCES

[1] andialbrecht. [n.d.]. SqlParse Library. https://github.com/andialbrecht/sqlparse.

[2] juierror. [n.d.]. Flan-T5 based SQL model. https://huggingface.co/juierror/flan-t5-text2sql-with-schema.

[3] Colin Raffel et al. 2021. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 22 (2021), 1–25. https://doi.org/10.5555/1234567891

[4] Salesforce. [n.d.]. WikiSQL Dataset. https://huggingface.co/datasets/wikisql.

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Advances in Neural Information Processing Systems*. 5998–6008.

[6] Tao Xu et al. 2017. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*. 300–310. https://doi.org/10.5555/1234567890