<u>**DOCUMENTATION**</u>

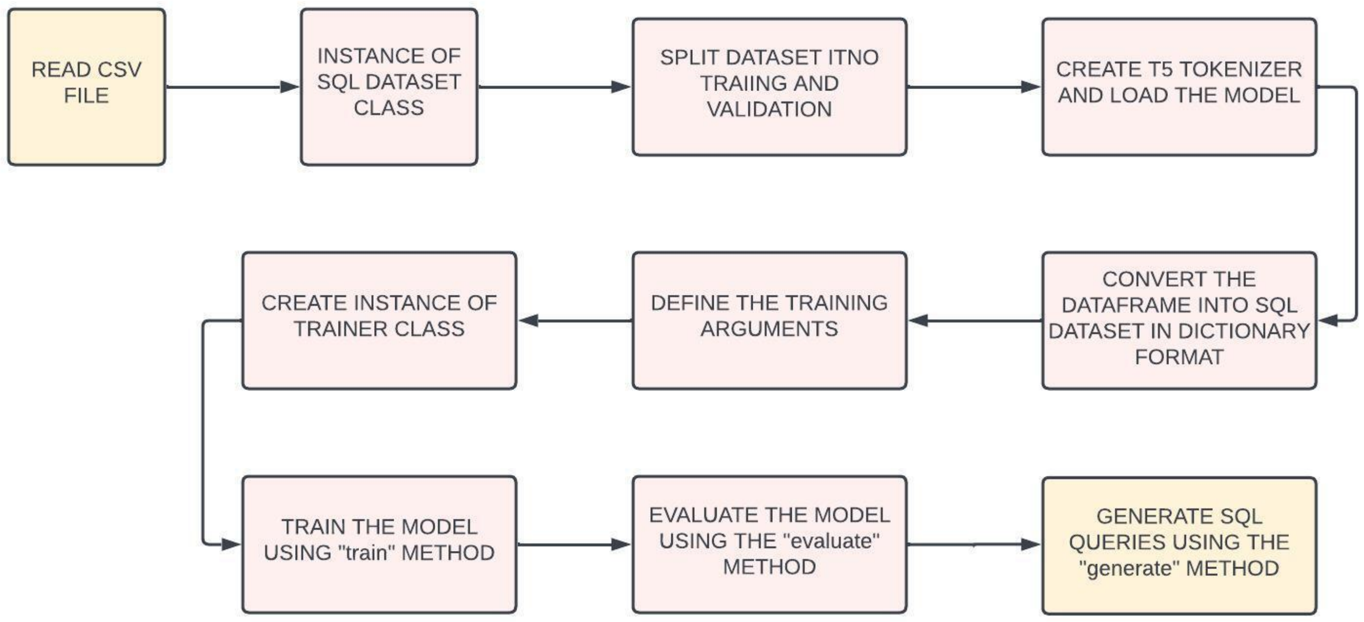
<u>NATURAL LANGUAGE STATEMENT TO SQL QUERIES</u>

DRAFT 3

TABLE OF CONTENTS

OVERVIEW AND FLOW OF CODE

The use case here is to develop the functionality to convert natural language statements to SQL queries. The goal of NLP to SQL is to make it easier for users to interact with databases by allowing them to use natural language queries rather than having to write complex SQL code. The dataset used in this is from a source like WikiSQL. This dataset contains natural languages and their corresponding SQL queries, which have been used to train the model. The pre-trained model that has been used is T5-base from hugging face. It is an appropriate model because it is designed to handle text-text transformations, making it suitable for converting natural language to SQL queries. Also, it is very easy to use, and the process of finetuning is also not complex.


READ CSV FILE
INSTANCE OF SQL DATASET CLASS
SPLIT DATASET ITNO TRAIING AND VALIDATION
CREATE T5 TOKENIZER AND LOAD THE MODEL
CONVERT THE DATAFRAME INTO SQL DATASET IN DICTIONARY FORMAT
DEFINE THE TRAINING ARGUMENTS
CREATE INSTANCE OF TRAINER CLASS
TRAIN THE MODEL USING "train" METHOD
EVALUATE THE MODEL USING THE "evaluate" METHOD
GENERATE SQL QUERIES USING THE "generate" METHOD

LIBRARIES USED –

Transformers library from hugging face – Library for Natural language processing tasks, it provides various pre-trained models that can be fine tuned further according to our use-case.

PyTorch – Library for deep learning and tensor computations.

Pandas – Library used for data manipulation and pre-processing of data.

Trainer – This class from the Transformers library is used for training the given model on the dataset.

Trainer Arguments – This class is used to define various parameters required to train the model.

DIFFERENT PARTS OF TOKENIZATION AND ENCODING -

Tokenization –

Tokenization is the process of splitting a text into smaller units called tokens. In the context of natural language processing (NLP), tokens are typically words, subwords, or characters. As it divides the raw text into meaningful chunks that machine learning models can process, tokenization is a crucial step in NLP tasks.

Token IDs –

In the encoding process, the text is converted into numerical representations, specifically token IDs, using a tokenizer. The token IDs are integers that map each token in the text to a unique numerical value. The specific values assigned to the tokens depend on the tokenizer and the underlying vocabulary it uses. For example, in our case, we are using a pre-trained tokenizer from the T5-base model.

Input Ids –

When we tokenize a sentence or a text, it is broken down into smaller units called tokens. Each token is assigned a specific numerical ID based on a predefined vocabulary, which is the pre-trained tokenizer in our case. These IDs are basically what is understandable to the model during training and are in machine readable format. The model learns to associate these input IDs with the desired outputs during training and can then perform text-to-text translation or other such tasks

Attention Mask –

During the process of tokenization, the input sentence may have variable lengths; however, when it is encoded or tokenized, the machine has to be given fixed length inputs. To handle this padding, tokens are added that don’t have any value but are just added to make all the sentences of fixed length.

The attention mask basically helps the model differentiate between the actual input id tokens and the padding tokens. Actual input id tokens are given the value of 1 and padding tokens value of 0. During training, the attention mask helps the model to focus only on the part which is relevant and disregard the padding tokens. It ensures that the model doesn't assign any importance or attention to the padding tokens, improving efficiency and accuracy.

Labels –

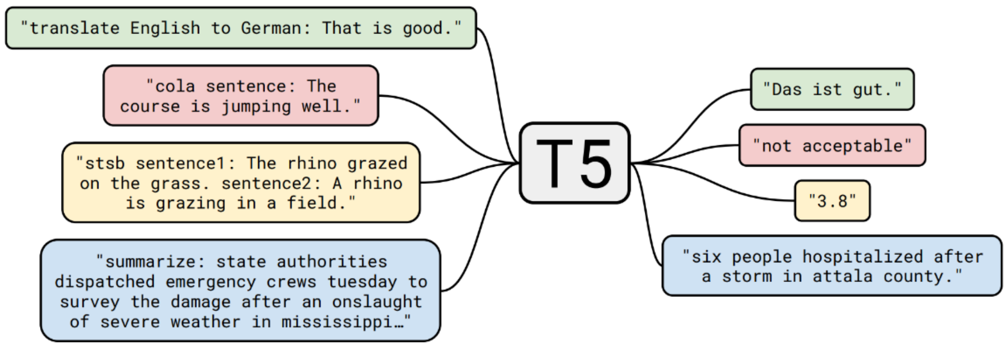
Labels are the target values which the model has to generate from natural language, in simple terms the natural language in the dataset is the source text and the SQL queries are the target text. The labels tensor contains the token IDs of the target SQL query after it has been tokenized using the same tokenizer. It serves as the ground truth during training and is used to compute the loss or error between the model's predicted token IDs and the actual token IDs of the SQL query.

WORKING OF MODEL T5

With T5, the researchers propose reframing all NLP tasks into a unified text-to-text-format where the input and output are always text strings, in contrast to BERT-style models that can only output either a class label or a span of the input. The T5 (Text-to-Text Transfer Transformer) model is a versatile transformer-based model that excels in various natural language processing (NLP) tasks.

In contrast to other famous Transformer based models like BERT or GPT, which is made up of either the encoder or decoder part of the Transformer, T5 paper showcases that using the complete encoder-decoder architecture is better than only using decoder. Apart from this, the paper also curated and released Colossal Clean Crawled Corpus (C4) - a huge crawled and cleaned dataset for pre-training language models using self-supervised learning.

Due to this nature of T5, for training or fine tuning, the model requires a pair of input and output sequences/text. Some example tasks that can be performed using T5 is shown below-


"translate English to German: That is good."
"cola sentence: The course is jumping well."
"stsb sentence1: The rhino grazed on the grass. sentence2: A rhino is grazing in a field."
"summarize: state authorities dispatched emergency crews tuesday to survey the damage after an onslaught of severe weather in mississippi…"
T5
"Das ist gut."
"not acceptable"
"3.8"
"six people hospitalized after a storm in attala county."

Working of the T5 model along with the Trainer API:

1. Tokenization: The input text is tokenized into smaller units (e.g., words, subwords, or characters) using the T5 tokenizer. The tokenizer converts the text into a sequence of tokens, and each token is assigned a specific numerical token ID based on a predefined vocabulary.

2. Encoding: The tokenized input text is fed into the T5 model, and the model's encoder processes the tokens to capture contextual representations of the input. These representations contain rich information about the meaning and structure of the text.

3. Trainer API Setup: To train the T5 model, the Trainer API is used. The API handles the training process, including data loading, batching, optimization, and evaluation.

4. Data Preparation: The input and output texts are prepared in a text-to-text format suitable for the T5 model. This means formulating the task as a text generation problem, where the input text serves as a prompt, and the output text is the target to be generated. The input-output pairs are tokenized and organized into training, validation, and test datasets.

5. Training Initialization: The T5 model is instantiated and combined with the Trainer API. The Trainer is configured with the model, training arguments (e.g., batch size, learning rate, number of epochs), and the training dataset.

6. Training Loop: The Trainer API handles the training loop. It iterates over the training dataset for the specified number of epochs. During each iteration, the model generates predictions based on the input prompts, and the generated outputs are compared with the target outputs to compute a loss. The Trainer uses an optimizer (e.g., Adam) to adjust the model's parameters and minimize the loss.

7. Evaluation: After each training epoch, the Trainer evaluates the model's performance on the validation dataset. This involves generating outputs for the validation prompts and comparing them to the target outputs to compute relevant metrics. The Trainer logs and tracks these metrics for monitoring the model's progress.

8. Saving and Loading: Once training is complete, the trained T5 model can be saved using the Trainer API's save methods. The saved model can be loaded later for inference or further fine-tuning

MODEL METRICS USED

```
train_runtime':      1461.7334,      'train_samples_per_second':      44.36,
'train_steps_per_second':    2.773,    'total_flos':    9871511884922880.0,
'train_loss': 0.39829802636655504, 'epoch': 1.0})
```

```
eval_loss':       0.13556520640850067,       'eval_runtime':       28.4914,
'eval_samples_per_second':   175.491,   'eval_steps_per_second':   10.986,
'epoch': 1.0}
```

1. Training Loss - It is used to measure the error or mismatch between the predicted output and the actual target output during the training phase.The goal is to minimize the training loss by adjusting the model's parameters through techniques like gradient descent or backpropagation.

2. `eval_loss`: This metric represents the average loss (error) calculated during the evaluation of the model. The loss is a measure of how well the model is performing in terms of the task it is trained for. Lower values indicate better performance.

3. `eval_runtime`: It is the total time taken for the evaluation of the model. It is typically measured in seconds and represents the time it took to process all the evaluation samples.

4. `eval_samples_per_second`: This metric indicates the number of evaluation samples processed per second. It is calculated by dividing the total number of evaluation samples by the evaluation runtime. Higher values indicate faster processing.

5. `eval_steps_per_second`: Similar to `eval_samples_per_second`, this metric represents the number of evaluation steps (iterations) performed per second. It is calculated by dividing the total number of evaluation steps by the evaluation runtime. Evaluation steps are usually smaller units of work within an evaluation loop.

PART - II

EXPANSION OF DATASET

The model was initially fine-tuned on WikiSql. Now, to improve its adaptability to different query types, data from the SparC and Spider datasets has been added. This expansion enables the model to handle diverse queries, making it more versatile and robust for natural language to SQL conversion tasks.

FURTHER PROBLEMS TO BE ADDRESSED

The current system utilizes a machine learning pre-trained model, T5, to convert natural language queries into SQL statements. However, a recurring issue arises where the table name in the generated SQL queries is consistently represented as "table." The objective is to enhance the system's capabilities by incorporating database connectivity and designing an approach that accurately identifies and includes the correct table name in the generated SQL queries.

APPROACHES RESEARCHED:

1. **Query Parsing:** Analyze the user query to identify keywords or patterns indicating the table name.
2. **Named Entity Recognition:** Use NER algorithms to extract table names from the user query.
3. **Database Schema Analysis:** Retrieve the schema information from the database and match keywords in the query with table names in the schema.
4. **Table Metadata Search:** Search for matching keywords or phrases in table metadata.

APPROACH USED:

The approach used is basically a culmination of all the approaches researched, compared to the previous iteration in which it was simply checking the table name with the generated query. In this approach, spaCy NLP technique is used to extract keywords from the user query, and it then connects to a MySQL database to fetch table and column names. It calculates Jaccard similarity between keywords and column names. Relevant columns are identified based on the similarity threshold. The most relevant table and column are chosen for the user's query.

STEPS INVOLVED:

1. Keyword Extraction: This part uses the SpaCy library to extract meaningful keywords from a user's input query. First, it loads the spaCy English model. Then, it processes the user query using the model, obtaining a linguistic analysis of the text. After that, it extracts nouns, proper nouns, and adjectives from the analyzed text based on their part-of-speech tags. Finally, it returns a list of these extracted keywords in lowercase form. All the stop words used in the English language, i.e., this, is, and of, are removed.
2. Connection Part: The code establishes a connection to a MySQL database located on the local host. It uses the mysql.connector library for this purpose. The connection parameters, such as the host, user, password, and database name, are specified.
3. Jaccard Similarity Part: The jaccard_similarity function calculates the Jaccard similarity between two sets. In this case, it computes the Jaccard similarity between the keywords extracted from the user query (using extract_keywords) and the individual column names of each table in the database.For each table, the code calculates the Jaccard similarity between the extracted keywords and the column names of that table. It does so by treating each column name as a set of words (splitting it into individual words) and comparing it with the set of keywords obtained from the user query.
4. The code starts by retrieving the names of all the tables in the MySQL database using the SQL command SHOW TABLES;.
5. It then iterates through each table and fetches the column names for each table using SHOW COLUMNS FROM {table_name};.
6. For each table, it calculates the Jaccard similarity between the extracted keywords and each column name using the jaccard_similarity function.
7. The highest similarity score and its associated column name are stored in the table_columns_mapping dictionary if the score exceeds a threshold value (0.1 in this case).
8. Additionally, it keeps track of the sum of similarity scores for each table in the similarity_sums dictionary.
9. It identifies the table with the highest similarity score and prints both the table name and the associated column name.
10. Finally, it replaces the correct table name in the generated query, logic used is that it replaces the word after the word “from” as in a SQL query all the table names are mentioned after the word from.

ADDITIONAL LIBRARIES USED -

- **spaCy -** spaCy is an open-source natural language processing library for Python. It is designed to be efficient, fast, and capable of handling large-scale text processing tasks. The library provides pre-trained models for various languages, including English. These models allow you to perform tokenization, POS tagging, named entity recognition, syntactic parsing, and other NLP tasks easily.

  In this code, the "en_core_web_sm" model is used, which is a small English model capable of handling most common NLP tasks. The token.pos_ attribute represents the part-of-speech tag of each token, and the list comprehension filters the tokens based on their POS tags to extract only meaningful keywords, including nouns, proper nouns, and adjectives.

- **mysql.connector:** This library is used to connect to the MySQL database and execute SQL queries.

PART - III

CHANGES AND ADDITIONS

1. **Replacing Jaccardian similarity with cosine similarity:** In this specific case, cosine similarity is likely the better choice for comparing the user's query (keywords) with the column names of tables. Since both are represented as binary vectors, cosine similarity effectively captures their similarity. It is particularly suited for continuous data with real-valued features and takes vector magnitude into account, which could be useful for this task.
2. **Making an API using Flask:** Creating an API using Flask involves using the Flask web framework in Python to build a web application that exposes specific endpoints to interact with the application's functionalities. An API was made for the machine learning model. On the webpage, the user enters the query, and the generated query is displayed along with the results from the corresponding table.

COSINE SIMILARITY

Cosine similarity is a measure of similarity between two non-zero vectors in a multi-dimensional space. It calculates the cosine of the angle between the two vectors and is commonly used to compare the similarity of documents or items in various applications.

Here's a brief explanation of cosine similarity:

1. **Vector Representation:** Assume we have two vectors, A and B, representing two items or documents. Each element of the vector corresponds to a specific feature or attribute, and the value of the element represents the importance or weight of that feature.

2. **Dot Product:** The first step is to compute the dot product of the two vectors. The dot product of two vectors is the sum of the products of their corresponding elements.

3. **Vector Magnitude:** Next, we calculate the magnitude (length) of each vector. The magnitude of a vector is computed by taking the square root of the sum of the squares of its elements.

4. **Cosine Similarity:** The cosine similarity is then obtained by dividing the dot product of the two vectors by the product of their magnitudes.

$$\text{Cosine Similarity} = (A \cdot B) / (|A| * |B|)$$

5. **Interpretation:** The cosine similarity ranges from -1 to 1. A similarity score of 1 indicates that the two vectors point in exactly the same direction, meaning they are identical. A score of -1 indicates that they point in opposite directions, implying complete dissimilarity. A score of 0 means the two vectors are orthogonal and have no similarity.

6. **Advantages:** One of the key advantages of cosine similarity is that it is independent of vector magnitude, meaning it only measures the direction of the vectors, not their length. This makes it useful when comparing documents or items of varying lengths.

7. **Applications:** Cosine similarity is widely used in information retrieval, text mining, natural language processing, recommendation systems, and collaborative filtering. It is especially helpful for tasks like document similarity, document clustering, and content-based filtering in recommender systems.

FLASK

Flask is a microweb framework for Python that allows you to build web applications quickly and with minimal boilerplate code. It is lightweight, flexible, and follows the WSGI (Web Server Gateway Interface) specification, making it easy to deploy on various web servers.

Important parts of Flask in brief:

1. **Routing:** Flask provides a powerful routing system using the @app.route decorator. You can define URL patterns for different views (endpoints) in your application, specifying the HTTP methods they handle (e.g., GET, POST).
2. **Views/Endpoints:** Views are Python functions that handle incoming requests and return responses. These views are associated with specific routes and perform actions based on the request data.
3. **Request and Response:** Flask's request object allows you to access data from incoming requests, such as form data, JSON, or query parameters. The response object helps you generate responses to be sent back to clients.
4. **Templates:** Flask integrates with Jinja2, a powerful templating engine. Templates allow you to separate the presentation layer from the application logic, making it easier to generate dynamic HTML content.
5. **Static Files:** Flask can serve static files (e.g., CSS, JavaScript) using the static folder in your project directory. This helps in organizing and serving static assets to the clients.