

1. 实验目的:

实现将自然语言转化为 SQL 的转化任务。（利用大模型转换：论文部分是 GPT-4，我们小组自行加了 glm-4 并实现数据清洗，看看国内大模型与 GPT4 在 text to SQL 预训练上的差距）

2. 实验设备: 三台笔记本电脑

3. 实验分工:

实验代码调试: 王嘉豪, 王彦军

评价标准: 王彦军

数据清洗: 王嘉豪, 王彦军, 吉嘉成

PPT 编写: 吉嘉成, 王彦军, 王嘉豪

实验总述: 王嘉豪

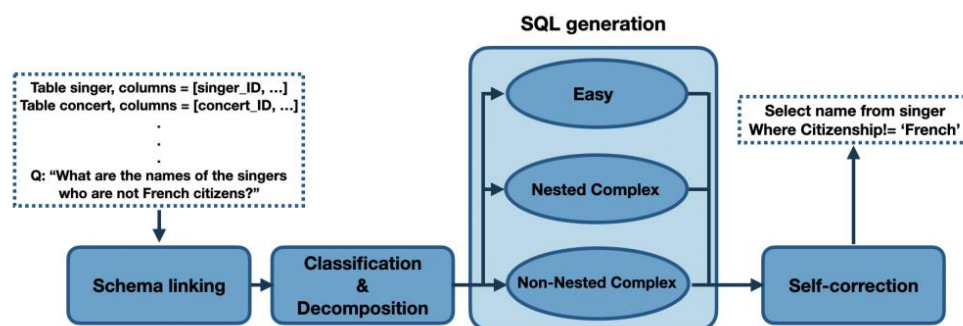
4. 实验内容:

(1) 任务综述:

我们研究将复杂的文本到 SQL 任务分解为更小的子任务的问题，以及这种分解如何显着提高大型语言模型 (LLM) 在推理过程中的性能。目前，在具有挑战性的文本到 SQL 数据集（例如 Spider）上，微调模型的性能与使用 LLM 的提示方法之间存在显著差距。我们证明 SQL 查询的生成可以分解为子问题，并且这些子问题的解决方案可以输入到 LLM 中以显着提高其性能。

(2) 主题理论部分:

基于这一思维过程，我们提出的分解文本到 SQL 任务的方法包括四个模块（如图 2 所示）：(1)模式链接，(2)查询分类和分解，(3) SQL 生成，以及(4)自校正，这些模块将在下面的小节中详细解释。虽然这些模块可以使用文献中的技术来实现，但我们都使用提示技术来实现它们，以表明如果将问题简单地分解到正确的粒度级别，llm 就能够全部解决它们。在提示中使用的少镜头示例是从各自的基准测试的训练集中获得的。

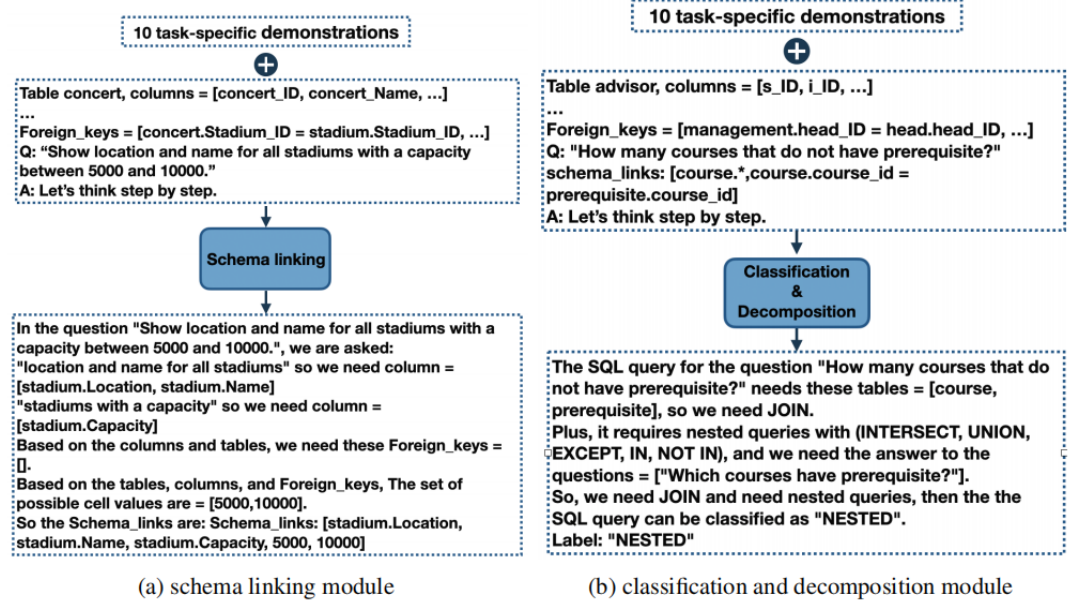


(1) seme link

模式链接负责识别在自然语言查询中对数据库模式和条件值的引用。它被证明有助于跨领域的通用性和复杂查询的综合[Lei 等人, 2020],

我们为模式链接设计了一个基于提示的模块。该提示包括从蜘蛛数据集的训练集中随机选择的 10 个样本。遵循思维链模板[Wei 等人, 2022b], 提示以“让我们一步一步地思考”开始，正如 Kojima 等人[2022]所建议的那样。对于问题中每次提到列名，将从给定的数据库模式中选择相应的列及其表。还从问题中

提取可能的实体和单元值。图 3a 说明了一个示例，完整的提示可以在附录 A.3 中找到。



(2) 4.2 Classification & Decomposition Module

对于每个连接，都有可能未检测到正确的表或连接条件。随着查询中连接数量的增加，至少有一个连接无法正确生成的机会也会增加。缓解这个问题的一种方法是引入一个模块来检测要连接的表。此外，一些查询具有程序组件，如不相关的子查询，它们可以独立生成并与主查询合并。

为了解决这些问题，我们引入了一个查询分类和分解模块。该模块将每个查询分为三个类之一：简单、非嵌套复杂和嵌套复杂。easy 类包括可以回答而无需连接或嵌套的单表查询。非嵌套类包括需要连接但不需要子查询的查询，而嵌套类中的查询可以包含连接、子查询和集操作。类标签对于我们的查询生成模块很重要，该模块为每个查询类使用不同的提示。除了类标签之外，查询分类和分解还检测非嵌套和嵌套查询的表集，以及嵌套查询的任何子查询。图 3b 显示了给模型的示例输入和模型生成的输出。

(3) SQL Generation Module

随着查询变得更加复杂，必须合并额外的中间步骤，以弥合自然语言问题和 SQL 语句之间的差距。这种差距在文献中被称为不匹配问题[Guo et al., 2019]，对 SQL 的生成提出了重大挑战，因为 SQL 主要是为查询关系数据库而设计的，而不是代表自然语言中的含义[Kate, 2008]。虽然更复杂的查询可以受益于在思维链式提示中列出中间步骤，但这样的列表可能会降低更简单任务的性能[Wei et al., 2022b]。在相同的基础上，我们的查询生成由三个模块组成，每个模块都面向不同的类。

对于我们简单课程中的问题，一个简单的没有中间步骤的少量提示就足够了。这个类的一个示例 E_j 的演示遵循<Q_j、S_j、A_j>的格式，其中 Q_j 和 A_j 分别以英语和 SQL 给出查询文本，S_j 表示模式链接。

我们的非嵌套的复杂类包括需要连接的查询。我们的错误分析(3)显示，在简单的少镜头提示下，找到连接两个表的正确列和外键来说是具有挑战性的，特别是当查询需要连接多个表时。为了解决这个问题，我们采用中间表示来弥

合查询和 SQL 语句之间的差距。在文献中已经介绍了各种中间表示形式。特别是，SemQL [Guo 等人，2019]删除了操作符 JOIN ON、JON、FROM 和组 BY，它们在自然语言查询中没有明确的对应项，并合并了有子句和 WHERE 子句。NatSQL [Gan 等人，2021]建立在 SemQL 的基础上，并删除了集操作符。自然语言查询中的表达式可能不会清楚地映射到唯一的 SQL 子句，或者它们可能会映射到多个子句，因此删除操作符会使从自然语言到 SQL 的转换更容易。作为我们的中间表示，我们使用 NatSQL，当与其他模型结合时，它显示出具有最先进的性能[Li 等人，2023a]。非嵌套复杂类的示例 Ej 的演示遵循<Qj、Sj、lj、Aj>的格式，其中 Sj 和 lj 分别表示第 j 个示例的模式链接和中间表示。

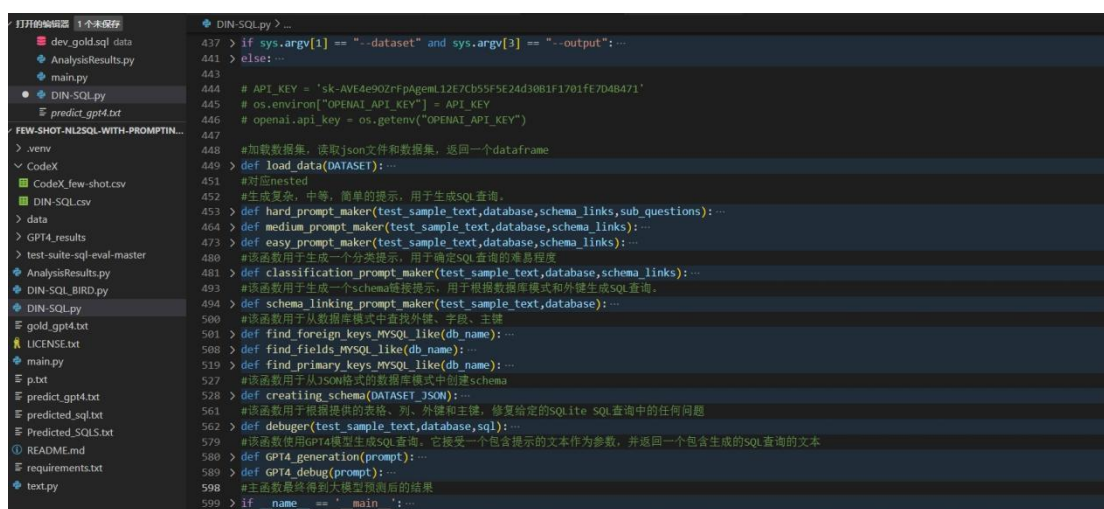
最后，嵌套的复杂类是最复杂的类型，在生成最终答案之前需要几个中间步骤。这个类可以包含不仅需要使用嵌套和集操作的子查询的查询，如除外、并集和相交，而且还需要多个表连接，与前一个类相同。为了将问题进一步分解为多个步骤，我们对这个类的提示符被设计为：LLM 应该首先解决从上一个模块生成的子查询，然后使用它们来生成最终答案。该类的提示符格式为<Qj、Sj、<Qj1、Aj1、...、Qjk、Ajk>、lj、Aj>，其中 k 表示子问题的数量，Qji 和 Aji 分别表示第 i 个子问题和第 i 个子查询。如前面所述，Qj 和 Aj 分别用英语和 SQL 表示查询，Sj 给出了模式链接，lj 是 NatSQL 的中间表示。

(4) Self-correction Module

生成的 SQL 查询有时可能有缺失或冗余的关键字，如 DESC、不同的和聚合函数。我们对多个 llm 的经验表明，这些问题在较大的 llm 中不太常见（例如，GPT-4 生成的查询比 CodeX 生成的查询有更少的 bug），但仍然存在。为了解决这个问题，我们提出了一个自修正模块，其中模型被指示来纠正这些小错误。这是在零镜头设置中实现的，其中只将有错误的代码提供给模型，并要求它修复 bug。我们为自我校正模块提出了两种不同的提示：通用的和温和的。通过通用提示，我们要求模型识别并纠正“buggySQL”中的错误。另一方面，温和的提示并不假定 SQL 查询有 buggy，而是要求模型检查任何潜在的问题，并对要检查的子句提供一些提示。

5. 实验过程：

(1) 代码部分主要函数及其功能：



```

437 > if sys.argv[1] == "--dataset" and sys.argv[3] == "--output": ...
441 > else: ...
443
444 # API_KEY = 'sk-AVE4e90ZrFpAgemL12E7Cb55FE24d30B1F1701fe7D48471'
445 # os.environ["OPENAI_API_KEY"] = API_KEY
446 # openai.api_key = os.getenv("OPENAI_API_KEY")
447
448 #加载数据集，读取json文件和数据集，返回一个dataframe
449 > def load_data(DATASET): ...
451 #对应nested
452 #生成复杂，中等，简单的提示，用于生成SQL查询。
453 > def hard_prompt_maker(test_sample_text,database,schema_links,sub_questions):...
454 > def medium_prompt_maker(test_sample_text,database,schema_links):...
455 > def easy_prompt_maker(test_sample_text,database,schema_links):...
456 #该函数用于生成一个分类提示，用于确定SQL查询的难易程度
457 > def classification_prompt_maker(test_sample_text,database,schema_links):...
458 #该函数用于生成一个schema链接提示，用于根据数据库模式和外键生成SQL查询。
459 > def schema_linking_prompt_maker(test_sample_text,database):...
460 #该函数用于从数据库模式中查找外键、字段、主键
461 > def find_foreign_keys_MySQL_like(db_name): ...
462 > def find_fields_MySQL_like(db_name): ...
463 > def find_primary_keys_MySQL_like(db_name): ...
464 #该函数用于从JSON格式的数据库模式中创建schema
465 > def creating_schema(DATASET_JSON): ...
466 #该函数用于根据提供的表名、列、外键和主键，修复给定的SQLite SQL查询中的任何问题
467 > def debugger(test_sample_text,database,sql): ...
468 #该函数使用GPT4模型生成SQL查询，它接受一个包含提示的文本作为参数，并返回一个包含生成的SQL查询的文本
469 > def GPT4_generation(prompt): ...
470 > def GPT4_debug(prompt): ...
471 #主函数是得到大模型返回后的结果
472 > if __name__ == '__main__': ...

```

(2) GPT-4 及 glm-4 大模型接口以及参数：

```

def GPT4_generation(prompt):
    response = openai.ChatCompletion.create(
        model="gpt-4",
        messages=[{"role": "user", "content": prompt}],
        n = 1,
        stream = False,
        temperature=0.0,
        max_tokens=600,
        top_p = 1.0,
        frequency_penalty=0.0,
        presence_penalty=0.0,
        stop = ["Q:"]
    )
    return response['choices'][0]['message']['content']

def GPT4_debug(prompt):
    response = openai.ChatCompletion.create(
        model="gpt-4",
        messages=[{"role": "user", "content": prompt}],
        n = 1,
        stream = False,
        temperature=0.0,
        max_tokens=350,
        top_p = 1.0,
        frequency_penalty=0.0,
        presence_penalty=0.0,
        stop = ["#", ";", "\n\n"]
    )
    return response['choices'][0]['message']['content']

```

```
response = openai.ChatCompletion.create(
```

```
    model="gpt-4",#指定使用的模型，这里是 gpt-4。
```

```
    messages=[{"role": "user", "content": prompt}],#对话历史，包括用户和 AI 的消息。
```

这里是传入一个包含用户消息的列表

```
    n = 1,#生成多少个回复，这里是 1
```

```
    stream = False,#是否流式传输回复，这里是 False，表示一次性返回所有回复。
```

`temperature=0.0`,#生成回复时的随机性，0.0 表示完全确定性的回复，1.0 表示完全随机的回复。

```
    max_tokens=600,#生成回复的最大长度，这里是 600。
```

```
    top_p = 1.0,#生成回复时考虑的前几个概率最高的词汇，1.0 表示只考虑概率最高的词汇。
```

```
    frequency_penalty=0.0,#对生成回复中频繁出现的词汇的惩罚，0.0 表示不惩罚。
```

```
    presence_penalty=0.0,#对生成回复中新出现的词汇的惩罚，0.0 表示不惩罚。
```

```
    stop = ["Q:"]#生成回复的停止条件，这里是遇到 "Q:" 就停止生成。
```

```
)
```

```
    return response['choices'][0]['message']['content']#返回对话窗口的所有内容
```

(3) 进行数据清洗

(4) 评价预测出的 SQL 与标准的 SQL 正确率

GPT4

:

```
● ⚡nicholas >> python .\evaluation.py --gold .\evaluation_examples\gold.txt --pred .\evaluation_examples\GPT4_predict.txt
--db .\database\ --table .\tables.json --etype exec
count          easy          medium          hard          extra          all
248            446            174            166            1034
===== EXECUTION ACCURACY =====
execution      0.923          0.874          0.764          0.627          0.828
LAPTOP-5R1DVIR5  D:\Desktop\test-suite-sql-eval-master  18.928s  9:19 AM
```

glm-4:

```
● ⚡nicholas >> python .\evaluation.py --gold .\evaluation_examples\gold.txt --pred .\evaluation_examples\glm4_predict.txt --db .\da
tabase\ --table .\tables.json --etype exec
count          easy          medium          hard          extra          all
248            446            174            166            1034
===== EXECUTION ACCURACY =====
execution      0.782          0.729          0.753          0.705          0.742
LAPTOP-5R1DVIR5  D:\Desktop\test-suite-sql-eval-master  18.993s  9:37 AM
● ⚡nicholas >>
```

6. 实验总结:

根据 GPT4 预测出的 SQL 结果来看, 大体实现了论文中的 80% 左右的正确率要求, 也成功证明 GPT4 在 text to sql 问题上的良好效果

附: 我们自行加入了 glm-4 大模型, 用同样的方法进行预测, 但由上图结果来看, 正确率相差 10%, 潜在的原因: (1) glm-4 的效果介于 gpt3.5-gpt4 之间 (2) glm-4 得到的是中文预测, 再将其转为英文的过程中出现了一些偏差, 所以效果不好。