探索式数据分析大作业实验报告

数据科学导论 2022-2023 春季学期

学院: ___信息学院__ 专业: __理科试验班__ 年级/班级: __21 级 2 班_____

姓名: _____罗思佳_____ 学号: __2021201679___

1. 引言

• 探索式数据分析的背景与意义

随着信息爆炸时代的到来，数据分析成为科学和工程领域中不可或缺的工具。在这个背景下，探索式数据分析扮演着重要的角色，它通过挖掘数据集中的模式和关联，帮助我们发现新的知识和见解。探索式数据分析的意义在于通过数据驱动的方法，揭示数据中隐藏的信息，指导决策、优化业务流程和改进产品设计。它提供了一种强大的工具，帮助我们从海量数据中提取有价值的洞察力，推动科学研究、商业决策和社会发展。

• 近似查询处理问题

在探索式数据分析中，近似查询处理问题是一个关键挑战。随着数据集规模的增大，执行准确的查询变得耗时且复杂，因此需要高效的近似查询方法。这涉及设计合适的索引结构和算法，以在大规模数据集上实现快速查询，同时平衡查询准确性和计算效率。本实验报告旨在介绍探索式数据分析中的近似查询处理问题，并阐述我们的解决方案。通过实验结果的描述和分析，展示我们的方法在大规模数据集上的有效性和可行性。

• 整体方案

使用采样的策略进行近似查询：

（1）采样阶段（aqp_offline）

对于每个列，使用采样方法对数据进行采样，生成样本集合(samples)。数值型列使用等间隔采样，非数值型列生成一个字典，包含每个类别及其出现的频次。计算每个数值型列的总和(col_sum)和平均值(col_avg)。

（2）查询处理阶段（aqp_online）

解析每个查询(Q)的 json 格式字符串为一个字典,存放在查询集合(queries)中。遍历每个查询，根据查询类型执行相应的处理函数。

（3）处理函数

对于 count 查询,根据查询的约束条件和采样信息计算出查询结果的近似值。如果查询中指定了 groupby 列，则返回每个类别的计数近似值；否则，返回总数据量的近似值。

对于 sum 查询，根据查询的约束条件和采样信息计算出查询结果的近似值。

如果查询中指定了 groupby 列，则返回每个类别的求和近似值；否则，返回总和的近似值。

对于 avg 查询，根据查询的结果列和采样信息返回平均值近似值。如果查询中指定了 groupby 列，则返回每个类别的平均值近似值；否则，返回总体的平均值近似值。

技术挑战：

(1)采样方法的选择：在 aqp_offline 函数中对数据进行采样，选择合适的采样方法对查询结果的准确性有重要影响。采样方法有伯努利采样、泊松采样、分层采样等等。我选择了简单的等间隔采样方法，但并不是对整个数据采样，而是按列进行采样，这样能与数据的特点相适应。

(2)采样量的确定：确定合适的采样量是一个挑战，过小的采样量可能导致估计结果不准确,而过大的采样量则会增加计算开销,同时也会影响在线查询的时间。为了找到合适的采样量，我尝试了多个数值，将误差和查询时间进行比较，确定较优的样本量。

• 实验结果

在 dlab 平台上测试时,查询负载Q1、Q2、Q3、Q4 的 msle 分别为0.000000、0.000049、0.063294、0.032585，在线时间均保持在 0.01s-0.14s 左右,离线时间为 70s 左右，表现较好。

2. 整体方案

- 问题定义

（1）近似查询处理问题的形式化定义如下：

给定一个包含大量数据的数据集 D，以及一组查询 Q，其中每个查询包含结果列 result_col、约束条件 predicate 和分组列 groupyby。查询的目标是计算在数据集 D 上满足约束条件的数据子集的结果列的近似值。

• result_col：结果列，一个列表（list），有若干项，每项可以是：

表中列，一个仅有 1 项的 list：[一个字符串表示列名]。若存在，则 groupby 列一定不为 "_None_"，且列名一定与 groupby 列名相同.

聚合函数，一个有 2 项的 list：[一个字符串表示聚合函数，一个字符串表示聚合函数作用列名]，如果聚合函数是 count 则作用列必为"*"

• predicate：查询的约束条件，一个 list 有若干项，每项是一个 dict，包含键值 col 表示约束的列名、lb 表示约束的下界、ub 表示约束的上界，若上界或下界没有约束则为字符串"_None_"

• groupyby：分组列名，若没有 groupby 则为字符串"_None_"

• ground_truth：答案（ground_truth），一个 list，其中每项是又一个 list，表示 ground_truth 中某一行，该 list 中每个项表示 ground_truth 中某行某列的值

（2）符号定义：

数据集：D

查询集合：Q = {q1, q2, ..., qn}，其中 qi 表示第 i 个查询

数据集 D 的列集合：C = {c1, c2, ..., cm}，其中 ci 表示第 i 列

数据集 D 的行数：N

数值型列集合：num_cols，包含数据集 D 中的数值型列

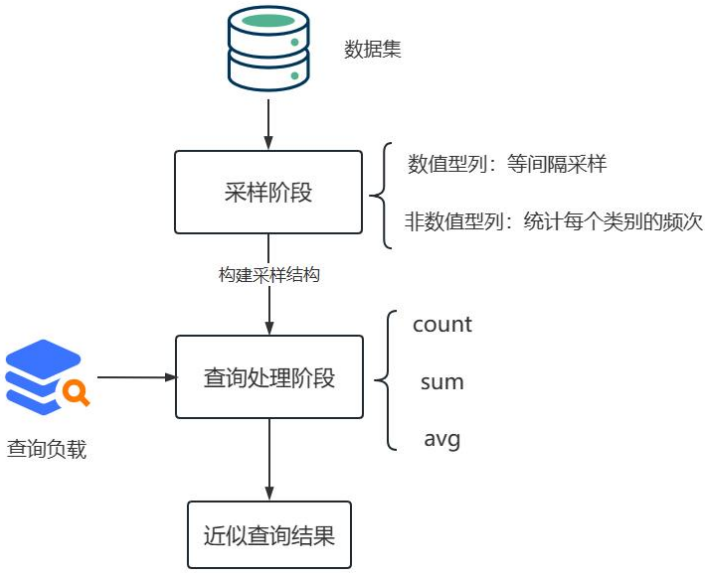
类别型列集合：cate_cols，包含数据集 D 中的类别型列

采样样本集合：samples，包含每个列的采样样本

数值型列的总和字典：col_sum，包含数值型列的总和

数值型列的平均值字典：col_avg，包含数值型列的平均值

- 整体框架



- 模块设计

采样阶段：

从数据集 D 中对每个列进行处理，生成采样样本集合 samples。数值型列排序后等间隔采样，非数值型列生成一个字典，包含每个类别及其出现的频次。计算每个数值型列的总和(col_sum)和平均值(col_avg)。

查询处理阶段：

①解析查询集合 Q 中的每个查询 qi。

②根据 qi 的结果列的聚合函数类型（count、sum、avg）、约束条件和分组列，调用不同的函数,使用采样样本集合 samples 进行近似计算。

近似查询结果：

返回近似查询结果，该结果是每个查询的近似值。

3. 详细设计

- 模型的符号化定义、推导、定理或引理等

➢数据集的符号化定义：

数据集 D：D = {r1, r2, ..., rN}，其中 ri 表示第 i 行的数据记录

数据记录 ri：ri = {ci1, ci2, ..., cim}，表示第 i 行的 m 个列的取值

➢查询符号化定义：

查询 qi 的约束条件：qi.predicate = [p1, p2, ..., pk]，其中 pi 表示约束条件中的第 i 个子条件

子条件 pi 的符号化定义：pi = {col, lb, ub}，表示约束条件中的列名、下界和上界

查询 qi 的结果列：qi.result_col = [[agg_func, col]] 或 [[col1],[agg_func,col]]，表示查询结果的列名和聚合函数

➢构建采样结构（aqp_offline）：

采样方法（sampling）：对数据集 D 的特定列 col_name 进行采样，得到采样样本 samples[col_name]。对于数值型列（num_cols）,先进行升序排序，然后等间隔采样（step=1000），存为一个列表；对于非数值型列（cate_cols），生成一个字典，包含每个类别及其出现的频次。计算每个数值型列的总和(col_sum)和平均值(col_avg)。

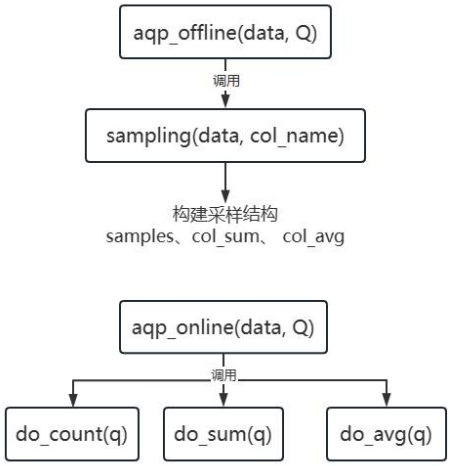
➢处理函数：

count 查询处理函数（do_count）：根据查询的约束条件和采样样本计算 count 聚合函数的近似值。

sum 查询处理函数（do_sum）：根据查询的约束条件和采样样本计算 sum 聚合函数的近似值。

avg 查询处理函数（do_avg）：根据查询的结果列和采样样本计算 avg 聚合函数的近似值。

- 算法的流程图，即不同函数是如何组织和调用的



- 关键函数的伪代码（注意是伪代码而非实际的 python 代码）

➢ 抽样函数

```
function sampling(data, col_name):
    sample_list ← []
    if col_name ∈ num_cols then //数值型列
        col_list ← data[col_name]
        Sort(col_list)              //升序排序
        for i ∈ range(0, total_num, step) do //等间隔采样
            sample_list.append(col_list[i])
        sample_list.append(col_list[-1])
        return sample_list
      else                          //非数值型列
        dic ← {}
        types ← list(set(data[col_name]))
        for type ∈ types do     //统计每个类别的频次
            dic[type] = list(data[col_name]).count(type)
        return dic
```

➢ count 聚合函数

```
function do_count(q):
    res←[]
    pred←q['predicate'] //取约束列
```

```
percent←1.0          // 比率初始化为 1
/*    计算满足约束条件的数据占总数据的比率     */
for p∈pred do
   if p['col']∈cate_cols then      //约束条件是非数值列
      num←samples[p['col']][p['lb']]      //取其频次
      percent←percent * (num / total_num) //乘以频率
   Else                                  //约束条件是数值列
      strip←samples[p['col']]             //取该列
      lower ← max(strip[0], p['lb'])    // 计算上下限
      upper ← min(strip[-1], p['ub'])
      num ← 0
      for i ∈ strip do
          if i >= lower and i <= upper then
              num ← num+1 //计算样本中满足约束条件的个数
      num ← num * ( total_num / sample_num) //乘以抽样间隔
      percent←percent * (num / total_num)
if len(pred) != 1 then //有多个约束条件
     percent←percent**0.855          //开方处理进行微调
/*       判断是否需要分组，计算最终结果        */
if q['groupby'] = '_None_' then
   res.append([total_num*percent])
else
   type_name ← q['result_col'][0][0]    //获取分组列名
   dic ← samples[q['result_col'][0][0]] //获取列名对应的字典
   for key ∈ dic.keys() do              //遍历
       res.append([key, dic[key] * percent])

return res
```

- sum 聚合函数

```
function do_sum(q):
```

```
res←[]
pred←q['predicate'] //取约束列
percent←1.0        // 比率初始化为1
/*    计算满足约束条件的数据占总数据的比率    */
for p∈pred do
   if p['col']∈cate_cols then      //约束条件是非数值列
      num←samples[p['col']][p['lb']]      //取其频次
      percent←percent * (num / total_num) //乘以频率
   Else                                //约束条件是数值列
      strip←samples[p['col']]            //取该列
      lower ← max(strip[0], p['lb'])    // 计算上下限
      upper ← min(strip[-1], p['ub'])
      num ← 0
      for i ∈ strip do
          if i >= lower and i <= upper then
              num ← num+1//计算样本中满足约束条件的个数
      num ← num * ( total_num / sample_num) //乘以抽样间隔
      percent←percent * (num / total_num)
if len(pred) != 1 then //有多个约束条件
    percent←percent**0.855          //开方处理进行微调
/*       判断是否需要分组，计算最终结果        */
certain_sum ← col_sum[q['result_col'][-1][1]]
if q['groupby'] = '_None_' then
    res.append([certain_sum*percent])
 else
    type_name ← q['result_col'][0][0]    //获取分组列名
    dic ← samples[q['result_col'][0][0]] //获取列名对应的字典
    for key ∈ dic.keys() do              //遍历
       res.append([key,certain_sum*percent*(dic[key]/total_num])

 return res
```

- avg 聚合函数

```
function do_avg(q):
    res ← []
    avg ← col_avg[q['result_col'][-1][1]]  //获取对应列的平均值
    if q['groupby'] = '_None_' then
        res.append([avg])
    else
        dic ← samples[q['result_col'][0][0]] //获取列名对应的字典
        for key ∈ dic.keys() do              //遍历
            res.append([key,avg])

    return res
```

- **通过一个例子端到端地展示上述详细设计**

以一个查询负载为例：

{"result_col": [["count", "*"]], "predicate": [{"col": "AIR_TIME", "lb": 822.9586581868816, "ub": 1521.2413418131182}], "groupby": "_None_", "ground_truth": [[998951]]}

• 先调用 aqp_offline 函数，构建采样结构，数值列排序后等间隔采样，并计算 sum 和 avg，非数值列统计各类别的频次

• 再调用 aqp_online 函数，因为聚合函数是 count，所以调用 do_count 函数。

• 这里的约束条件是 AIR_TIME 处于 822.9586581868816 和 1521.2413418131187 之间，AIR_TIME 是数值型列，所以遍历 sample['AIR_TIME']中的样本数据，统计有多少个数据满足约束条件，然后乘以采样的间隔，估计总数据集中满足约束条件的数据量，除以总数据量得到比率 percent。因为 groupyby 为 None，不需要分组，所以直接将 total_num*percent 添加到结果列表中。

• eval_result.json 中的结果为 {"output": [[998000.0]], "Loss": 1.7110192529175193e-07},误差非常小。

4. 实验结果及分析

- 实验设置：数据集描述、查询特点描述、实验环境等

（1）数据集描述：

数据集为一个关系数据表，来自[IDEBench]生成的与航班有关的数据

数据共包含 1,000,000 行和 12 列，其中包含 7 个数值型（Numerical）的数据列和 5 个类别型（Categorical）的数据列

（2）查询特点描述

提供四组查询负载，分别表示为 Q1 - Q4

* Q1：结果列仅有表中列和 count*，仅有 1 个筛选条件

* Q2：结果列仅有表中列和聚合函数 avg，不超过 2 个筛选条件

* Q3：结果列有表中列和聚合函数 count、avg，sum，若干筛选条件

* Q4：结果列有表中列和聚合函数 count、avg，sum，分组列仅作用于字符串类型的列且有分组列的 query 仅有 1 个筛选条件，若无分组列则有若干筛选条件

（3）实验环境

使用 dlab 的在线环境，CPU 为 8C16G

- 整体实验结果：
  - 四个查询负载上的实验结果

表 1 整体实验结果

查询负载	误差（MSLE）	在线时间	离线时间
Q1	0.000000	0.097466	70.119721
Q2	0.000049	0.011373	70.091155
Q3	0.063294	0.081011	70.059758
Q4	0.032585	0.125337	70.695641

误差（MSLE）：对于每个查询，计算了模型的均方对数误差。根据结果，不同查询的误差（MSLE）在 0 到 0.033 之间，表示模型的预测结果与真实值之间的平均相对误差较小。

在线时间：对于每个查询，记录了模型处理查询的在线时间。在线时间在 0.011 到 0.13 之间，表示模型能够在较短的时间内对查询进行处理。

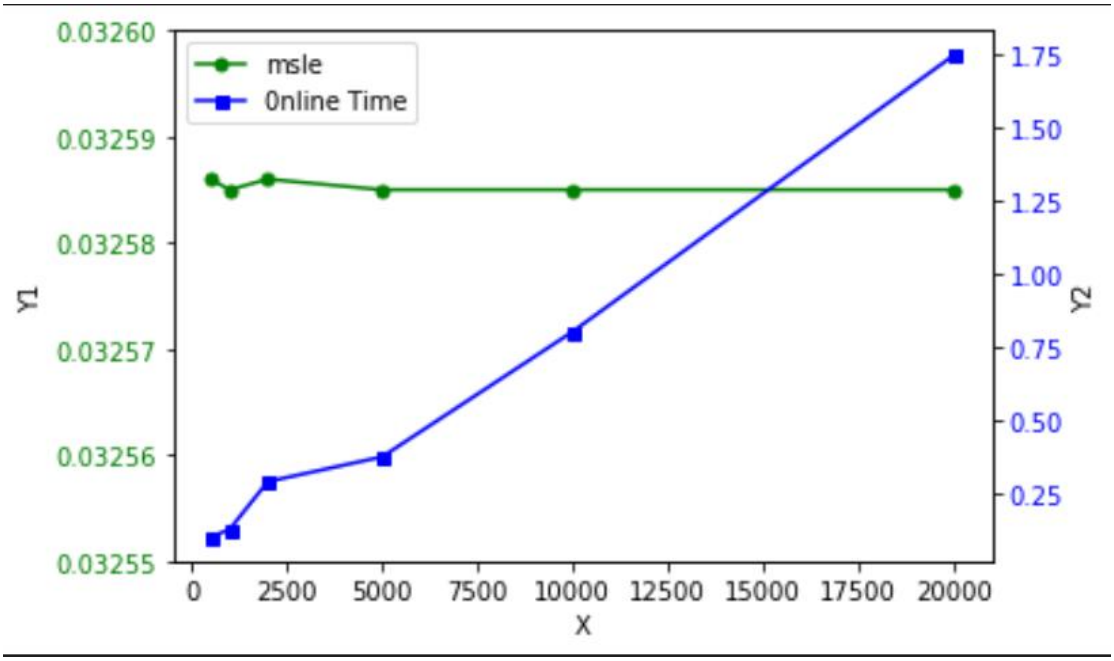
离线时间：记录了模型构建过程的离线时间，该过程包括采样结构的构建。离线时间保持稳定，保持在 70s 左右。

综上，算法在不同查询负载下表现出较低的误差和快速的在线处理时间。离线构建阶段的时间相对较长，但在实验中保持稳定。

- 详细实验分析：
  - 参数分析

因为代码中涉及到的主要参数是样本量 sample_num，对样本量进行改变，从 500 到 2000，绘制 Q4 的折线图：

样本量	误差	在线时间
500	0.032586	0.099472
1000	0.032585	0.125337
2000	0.032586	0.288449
5000	0.032585	0.373144
10000	0.032585	0.801159
20000	0.032585	1.745779



可以看出，误差基本保持不变，在线查询时间随样本量的增加而变长。

5. 总结及体会

遇到的问题：

（1）最开始我是对整体进行简单随机抽样（最 naive 的方法），这样不仅没有充分利用数据的特点，估算出来的值也不够准确。因为数据包括数值型列和类别型列，可以分开处理，对数值型列建立一个存有该列数据的列表，对类别型列建立一个存有各个属性对应的频次的字典。

（2）在估算 count 和 sum 时，我是先分别计算满足每个约束条件的列的数据占总数据的比率，然后再乘起来，得到满足所有约束条件的数据占总数据的比率，

但这样没有考虑交叉的情况，即某一数据行可能同时满足多个约束条件，导致算得的比率偏小。因为在 offline 函数中我是对列分开统计的，所以无法得知有多少数据行是这种情况，因此我对最后总的比率进行 0.855 的幂次处理，进行了略微放大，从而将 msle 误差减小了一些。

心得体会

这次实验让我感受到了探索式数据分析在现代科学和工程领域中的重要意义。随着数据的不断增长和积累，我们需要从海量数据中提取有价值的信息和见解，以指导决策和推动发展。近似查询处理作为探索式数据分析中的关键问题，具有重要的研究价值和实际应用意义。

我也了解到了许多近似查询处理的策略，如采样、直方图、Sketches 等等。在本次实验中，我采用了采样的策略进行近似查询处理，通过对数据进行合适的采样和统计，实现了在大规模数据集上的快速查询。这一解决方案不仅提高了查询的效率，还能够在一定程度上保持查询结果的准确性。但我也知道这只是近似查询处理领域的一个初步探索，仍然存在许多可以改进和优化的方面，例如使用不同的采样策略、使用直方图的策略等等。

参考文献

[1]胡欢,李建中.基于随机抽样的近似聚集查询处理综述[J].智能计算机与应用,2022,12(06):166-169.

[2]Yi Jun Wang,Han Hu Wang,Hui Li. A Histogram Based Analytical Approximate Query Processing for Massive Data[J]. Applied Mechanics and Materials,2013,2700(411-414).

[3]Philipp Rösch,Wolfgang Lehner. Optimizing Sample Design for Approximate Query Processing[J]. International Journal of Knowledge-Based Organizations (IJKBO),2013,3(4).

[4]Surajit Chaudhuri,Gautam Das,Vivek Narasayya. Optimized stratified sampling for approximate query processing[J]. ACM Transactions on Database Systems (TODS),2007,32(2).

附录：

这部分粘贴所有的源代码，格式如下：

代码文件：aqp.py

粘贴代码

```
import pandas as pd
import json

samples = {}
col_sum = {}
col_avg = {}

columns = [
    'YEAR_DATE', 'UNIQUE_CARRIER', 'ORIGIN', 'ORIGIN_STATE_ABR', 'DEST',
    'DEST_STATE_ABR', 'DEP_DELAY', 'TAXI_OUT', 'TAXI_IN', 'ARR_DELAY',
    'AIR_TIME', 'DISTANCE'
]

num_cols = [
    'YEAR_DATE', 'DEP_DELAY', 'TAXI_OUT', 'TAXI_IN', 'ARR_DELAY', 'AIR_TIME',
    'DISTANCE'
]
cate_cols = [
    'UNIQUE_CARRIER', 'ORIGIN', 'ORIGIN_STATE_ABR', 'DEST', 'DEST_STATE_ABR'
]

total_num = 1000000  # 总数据量
sample_num = 1000  # 样本量


def aqp_online(data: pd.DataFrame, Q: list) -> list:
    global columns, samples, col_sum, col_avg, num_cols, cate_cols
    queries = []  # 把 Q 的每一条 json 格式字符串，解析成一个 dict
    for i in Q:
        queries.append(json.loads(i))

    results = []  # 存放每条 query 的 aqp 结果
    for q in queries:
        result_col = q['result_col']
        if result_col[-1][0] == 'count':
            results.append(do_count(q))
        elif result_col[-1][0] == 'avg':
            results.append(do_avg(q))
        elif result_col[-1][0] == 'sum':
            results.append(do_sum(q))
```

```
            # print(results[-1])
            results[-1] = json.dumps(results[-1], ensure_ascii=False)

    return results


def aqp_offline(data: pd.DataFrame, Q: list) -> None:
    '''无需返回任何值
    必须编写的 aqp_offline 函数，用 data 和 Offline-workload，构建采样、索引、机器学习相关的结构或模型
    data 是一个 DataFrame，Q 是一个包含多个 json 格式字符串的 list
    如果你的算法无需构建结构或模型，该函数可以为空
    '''
    global columns, samples, col_sum, col_avg, num_cols, cate_cols

    # 采样
    for col in columns:
        samples[col] = sampling(data, col)
    # 求每个数值型列的 sum 和 avg
    for col in num_cols:
        col_sum[col] = data[col].sum()
        col_avg[col] = data[col].mean()


def sampling(data, col_name):
    global columns, samples, num_cols, cate_cols, total_num, sample_num

    sample_list = []
    if col_name in num_cols:  # 是数值型列，等间隔采样
        col_list = list(data[col_name])
        col_list.sort()  # 排序
        step = int(total_num / sample_num)
        for i in range(0, total_num, step):
            sample_list.append(col_list[i])
        sample_list.append(col_list[-1])
        return sample_list
    else:  # 非数值型列，统计频次
        dic = {}
        types = list(set(data[col_name]))
        for type in types:
            dic[type] = list(data[col_name]).count(type)
        return dic
```

```
def do_count(q):
    res = []
    pred = q['predicate']
    percent = 1.0

    # 计算百分比
    for p in pred:
        if p['col'] in cate_cols:
            num = samples[p['col']][p['lb']]
            percent *= num / total_num

        else:
            lst = samples[p['col']]
            # interval = float((lst[-1] - lst[0])) / sample_num
            lower = 0.0
            upper = 0.0
            if p['lb'] != '_None_':
                lower = max(lst[0], p['lb'])
            else:
                lower = lst[0]

            if p['ub'] != '_None_':
                upper = min(lst[-1], p['ub'])
            else:
                upper = lst[-1]

            num = 0
            for i in lst:
                if i >= lower and i <= upper:
                    num += 1

            num *= total_num / sample_num
            percent *= num / total_num
    if len(pred) != 1:
        percent = percent**0.855   # 微调
    if q['groupby'] == '_None_':
        res.append([total_num * percent])

    else:
        type_name = q['result_col'][0][0]
        for i in pred:
            if i['col'] == type_name:   # 约束条件是类别列
                res.append([i['lb'], total_num * percent])
                return res
```

```
            dic = samples[q['result_col']][0][0]
            for key in dic.keys():
                res.append([key, dic[key] * percent])

    return res


def do_sum(q):
    res = []
    pred = q['predicate']
    percent = 1.0

    # 计算百分比
    for p in pred:
        if p['col'] in cate_cols:
            num = samples[p['col']][p['lb']]
            percent *= num / total_num

        else:
            lst = samples[p['col']]
            # interval = (lst[-1] - lst[0]) / sample_num

            lower = 0.0
            upper = 0.0
            if p['lb'] != "_None_":
                lower = max(lst[0], p['lb'])
            else:
                lower = lst[0]

            if p['ub'] != "_None_":
                upper = min(lst[-1], p['ub'])
            else:
                upper = lst[-1]

            num = 0
            for i in lst:
                if i >= lower and i <= upper:
                    num += 1

            num *= total_num / sample_num
            # num = (upper - lower) * step / interval
            percent *= num / total_num
    if len(pred) != 1:
```

```
            percent = percent**0.855    # 微调
        certain_sum = col_sum[q['result_col'][-1][1]]
        if q['groupby'] == '_None_':
            res.append([certain_sum * percent])

        else:
            type_name = q['result_col'][0][0]
            for i in pred:
                if i['col'] == type_name:    # 约束条件是类别列
                    res.append([i['lb'], col_sum[q['result_col'][1][1]] * percent])
                    return res
            dic = samples[q['result_col'][0][0]]
            for key in dic.keys():
                res.append([key, certain_sum * percent * (dic[key] / total_num)])

        return res


def do_avg(q):
        res = []
        avg = col_avg[q['result_col'][-1][1]]
        if q['groupby'] == '_None_':
            res.append([avg])
        else:
            dct = samples[q['result_col'][0][0]]
            for key in dct.keys():
                res.append([key, avg])

        return res
```