Lab1 Report

PB22111639 马筱雅

问题描述

- **问题背景知识：**
  对于一个数据表 `T` ，表 `T` 中可以通过结合外部信息识别表中记录的属性的最小集合称为准标识符属性集合。在表 `T` 中，对于任意准标识符属性值都相同的记录集合，称为一个等价类。若 `T` 的每个等价类大小至少为 `k` ，则称该表在该属性集上是k匿名的。 `Mondrian` 算法是一种使用贪心算法对数据进行k匿名的一种方式。将一个准标识符作为一个维度， `Mondrian` 算法通过对可划分维度的数据进行分割，再把得到的子集进行重复操作，直至所有维度都不能再划分，来实现 `K` 匿名。

- **问题要求：**
  分别通过基础型 `Mondrian` 算法和通用型 `Mondrian` 算法实现对adult.data原始数据集的k-匿名，基础型 `Mondrian` 算法将非数值型数据映射成数值型数据，通用型 `Mondrian` 算法采用泛化树处理非数值型数据。并且，需要分析 `Mondrian` 算法的性能，性能影响因子及算法思路和特点。具体为
  i. 补全 `anonymize_strict` 函数， `anonymize_relaxed` 函数， `split_categorical` 函数
  ii. 阐述 `Mondrian` 算法的解决思路并绘制流程图
  iii. 讨论全局损失、执行时间、安全性之间的关系
  iv. 总结Mondrian算法的特点

解决思路

基础型Mondrian算法

由于基础型 `Mondrian` 算法主要用来处理数值型数据，所以对于给定的数据集，首先对其非数值型数据进行简单映射，相同属性值映射成一个整数，且整数满足从0递增，从而编码成数值型数据。对于得到的数值型数据应用 `Mondrian` 算法进行划分。

具体解决顺序及细节如下：

- **准备工作**：对于一个数据集，首先对每个维度的数据进行统计，对不同的数据按从小到大的顺序重新编码，用QI_DICT字典记录，即QI_DICT[value]对应于该value在所有不同的有序value中的位置，并将所有维度都设置为可划分，从而方便后续划分。将一个数据集初始化为 `Partition` 类，包含 `allow` 代表着某个维度是否可以继续划分， `low` 代表着该数据集的最小值重编码后的索引，即在QI_DICT中的索引，同理 `high` ，这两个属性用来最终表示划分区间的上下界。 `allow` 初始化为[1]*QI_LEN，表示所有维度可划分，上述操作方便对数据集是否可继续划分进行判断。
- **找可划分维度**：如果数据集中存在某一维度的 `allow` 为真，并且满足该维最大值编码与最小值编码之差的绝对值小于等于初始数据集的数据编码后宽度，则选择该维度。这样做是出于若划分的子集最大编码与最小编码之差大于原数据集，则说明计算有误，该维度数据有错。

- **根据维度找中位数等信息**：若要找中位数，需要先计算该维数据中每一个数据出现的频率。中位数是对排序后的数据而言，所以对数据进行排序，得到不同有序数据对应的频率的字典。通过对字典进行从小到大遍历，找到所需数据值，假设小于该数据值的总数为 `M` ，所有数据个数为 `N` ，该数据值对应数量为 `X` ，使得 `M < N/2` 且 `M + X >= N/2` 。该数据值即为中位数。为了更新划分子集和原数据集的最大最小值，也需要同时找到紧邻中位数的右值和该维数据的最大最小值。
- **更新选择维度的上下界**：在找中位数时，同时返回了该维度的最大最小值，通过 `QI_DICT` 即可得到对应的上下界编码，从而对 `Partition` 的 `low` 和 `high` 进行更新。
- **根据中位数进行划分**：划分分为 `strict` 和 `relax` 两种方式。
  - **strict**：记录中位数对应的 `QI_DICT` 编号 `mean` ，则遍历数据集，若该维度数据在 `QI_DICT` 中的编号小于等于 `mean` ，则说明数据小于等于中位数，划分为左子集，否则划分为右子集，并根据划分的中位数及其右相邻数据位置更新得到的划分子集的上下界。
  - **relax**： 录中位数对应的 `QI_DICT` 编号 `mean` ，则遍历数据集，若该维度数据在 `QI_DICT` 中的编号小于 `mean` ，则说明数据小于中位数，划分为左子集，若大于，划分为右子集，对于等于的值，再次进行划分，分别添加到左右子集中，使两者数据数目相同或者相差1。并根据划分的中位数及其右相邻数据位置更新得到的划分子集的上下界。
- **判断是否满足k匿名**：若划分后的两个子集数目均大于等于 `k` ，则满足k匿名，否则不满足，说明该维度不可继续划分，把 `Partition` 类该维度对应的 `allow` 值置为假。
- **递归**：若上述操作得到了正确的中位数两个子集满足k匿名的条件，则对两个子集分别重复上述操作。若得到的两个子集不满足k匿名的条件，则寻找该数据集的下一个可分维度，重复上述操作，直至对一个数据集，所有维度都不可再进行划分，即得到k匿名后的结果。
- **计算NCP**：对于得到的划分，对每一个划分计算NCP并进行相加，最后计算GCP。

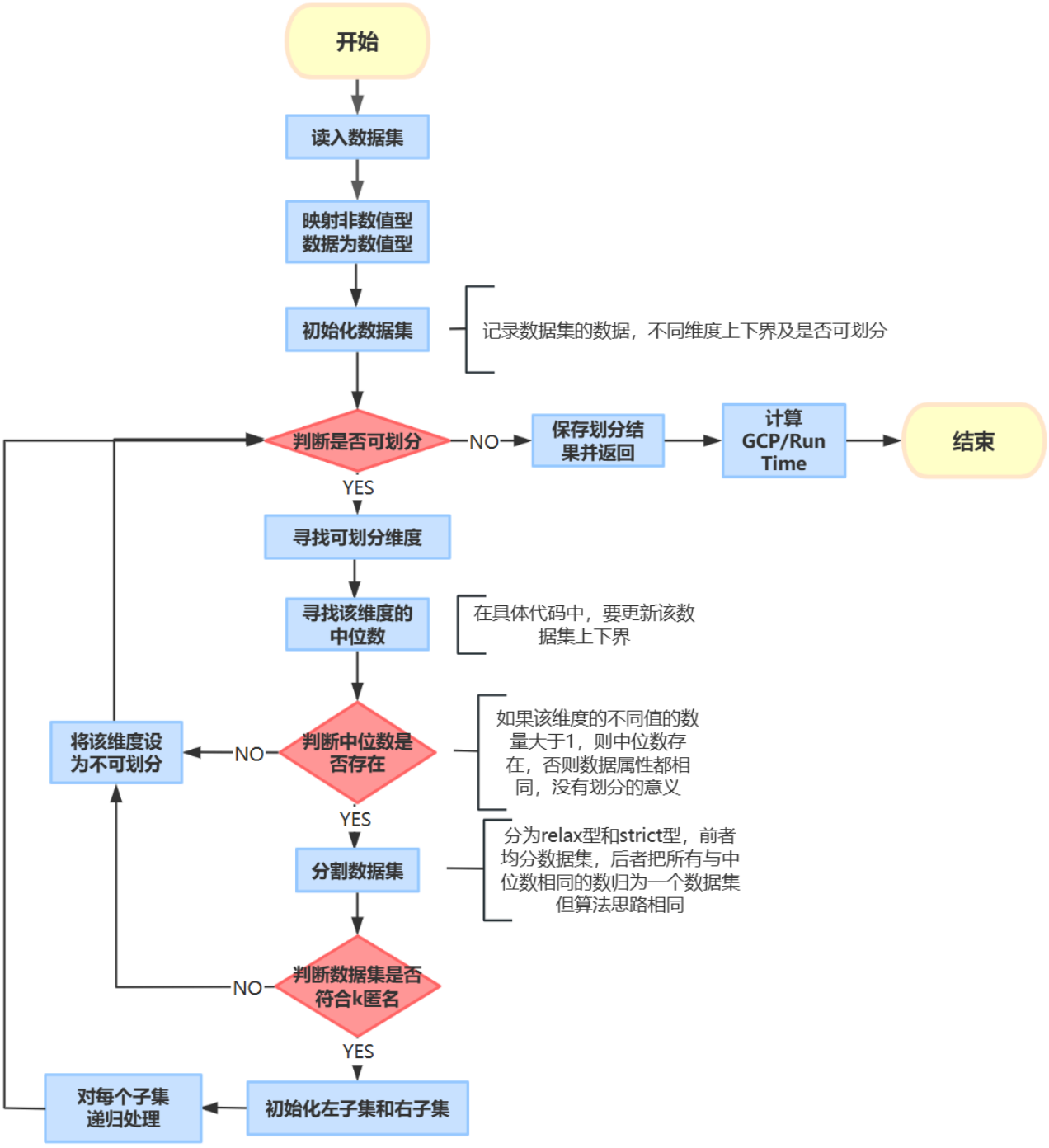
通用型Mondrian算法

通用型Mondrian算法与基础型Mondrian算法解决思路类似，不同的是对非数值型数据的处理方法。通用型Mondrian算法对非数值型数据进行泛化，在进行数据集划分时采用按照泛化树结构从上到下不断分解的方式。具体解决方法如下

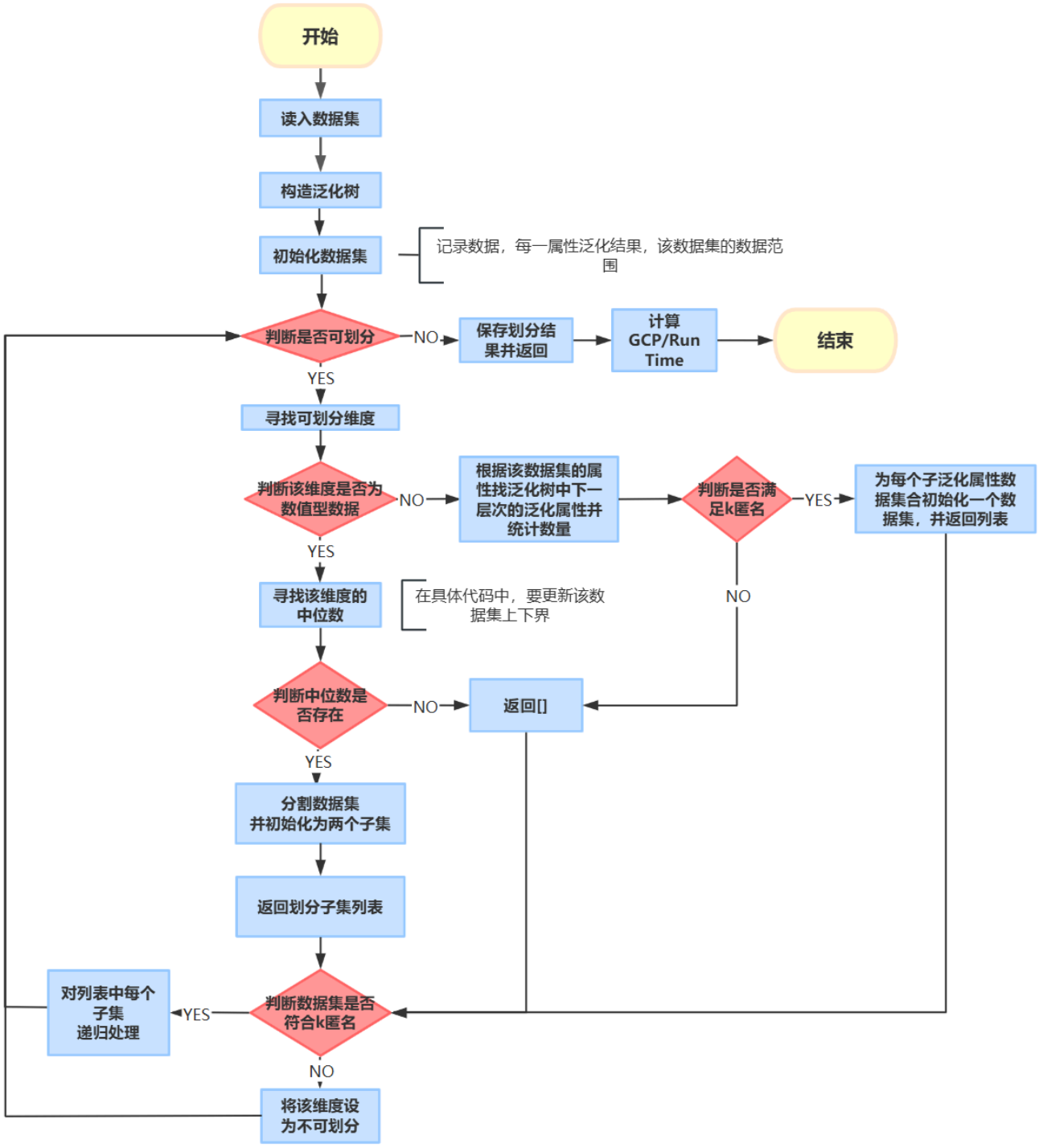
- **准备工作**：构造泛化树，读入数据。用QI_RANGE存放每个属性的范围，对于数值型属性，QI_RANGE存放数据的最大最小值之差，wtemp存放数据的按序索引范围，middle存储了上下界的值。对于非数值型属性，QI_RANGE存放了该属性的泛化树的总叶子结点个数， wtemp与QI_RANGE相同，middle存放最上层的泛化树结构。
- **找可划分为维度**：数值型处理方式与基础型Mondrian算法相同，对于非数值型，如果满足该维度属性的范围小于等于总的数据集范围，且该维度可划分，则可以进行选择。
- **划分**：
  - 对于数值型数据划分思路与基础型relax类似，即先找到中位数，根据中位数判断是否可划分，若可划分，返回划分后的类的列表，否则返回空列表。- 对于非数值型数据，根据泛化树，采用从上到下不断分割的方式。首先根据 `Partition` 类的 `middle` 属性，找到其子孩子结点，遍历该数据集，把该数据集根据属性划分到不同的子孩子结点对应的数据集中。若满足k匿名，则返回得到的数据类的列表，否则返回空列表。
- **判断是否满足k匿名**：若不满足k匿名，则说明该维度不可划分，继续寻找可划分的下一维度，若可划分，则对划分得到的每个数据集分别进行递归，重复找可划分维度之后的所有操作。
- **结束条件**：一个数据集的所有维度均不可继续划分
- **计算NCP**：对于得到的划分，对每一个划分计算NCP并进行相加，最后计算GCP。

算法流程图

基础型Mondrian算法



通用型Mondrian算法



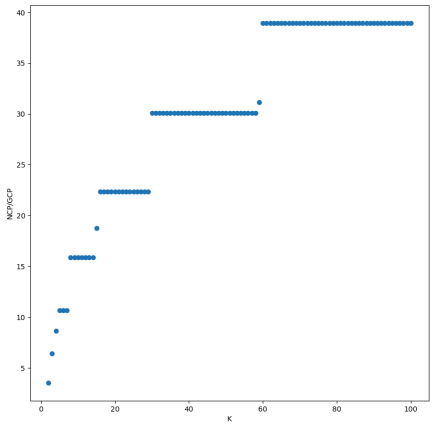
影响因子分析

- 影响因素： `K` ， `qi` , `data_size`
- 被影响因素： `GCP` ， `RunTime` 。根据代码分析，代码计算的 `NCP` 即为 `GCP`
- 分析方法：**控制变量法**。为了方便记录不同的值，将每次划分时的 `K` ， `qi` ， `data_size` 及其生成的 `NCP/GCP` 和 `RunTime` 存储到一个 `csv` 文件中。

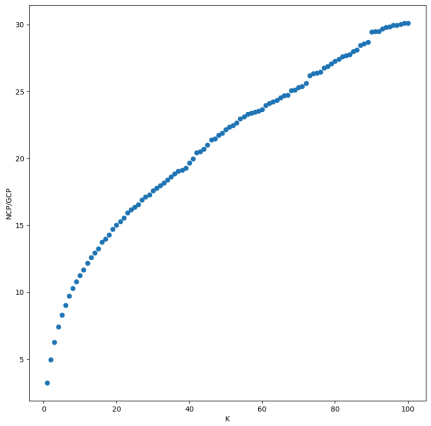
  注：图中NCP/GCP的单位为%，RunTime单位为s

  注：在以下的报告中，从公式等角度对为什么产生如此结果进行了分析，所以稍显繁琐

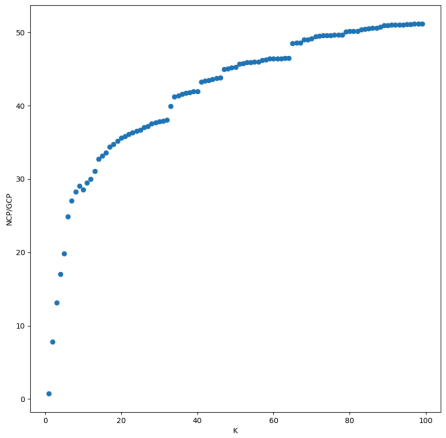
K对GCP的影响



relax:K对GCP的影响



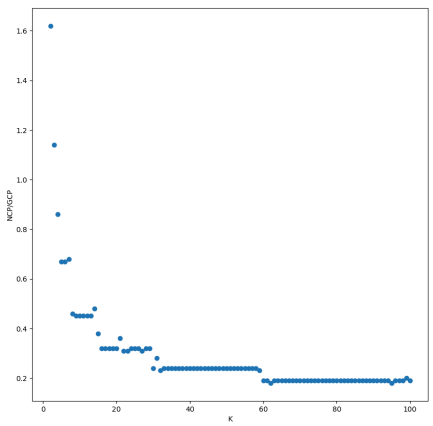
strict:K对GCP的影响



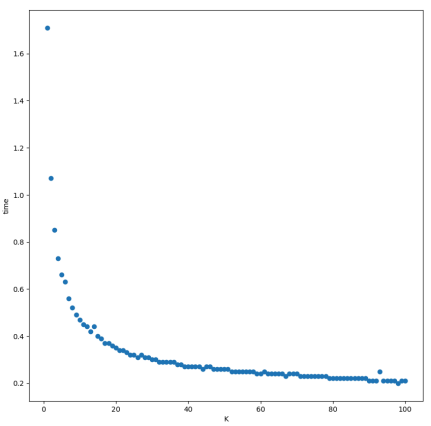
Mondrian2:K对GCP的影响

- **分析方法**：保持 `qi` 和 `data_size` 不变，使 `k` 的值从 `1-100` 递增，记录 `NCP/GCP` ，画出散点图。
- **结果**：通过三个图可以看出，随着 `K` 的增大， `GCP` 变大，且变大速率越来越慢。对于 `relax` 方式，存在相邻的 `k` 对应的 `GCP` 相同，但总体仍呈增大趋势。
- **原因**：
  - 对于数值型数据，$NCP_{A_i}t = \frac{z_i - y_i}{|A_i|}$，其中$|A_i|$是第 `i` 维数据的范围，$z_i - y_i$是经 `Mondrian` 算法划分得到的某一子集的范围。当数据总数相同时， `k` 越大，对应的$z_i - y_i$越大，则该维度对应的 `NCP` 相应会增大，最终计算得到的 `GCP` 是所有数据 `NCP` 之和的平均，也就相应增大。
  - 对于非数值型数据， `k` 越大代表着泛化结果越宽泛。根据公式$NCP_{A_i}t = \frac{|u|}{|A_i|}$ ，其中$|u|$对应该数据泛化结果 `u` 在泛化树中对应的子树中叶子节点个数。则 `k` 越大，对应的泛化后的结果在泛化树中越靠近根节点，则对应的叶子结点数量越多，则相应该条数据对应的 `NCP` 增大，以此类推，平均的 `NCP` ，即 `GCP` 也就增大。
  - 对于 `relax` 型分割方式，每次分割进行均分，直到不可分割。故对于 `k` ，设某次分割后的数据集大小为 `m` ，若该数据集不能继续分，则 `m < 2k` 且 `m >= k` 。由于对于不同的 `k` ，都是采用相同的方式，只是划分结束时间不同，因此，对于某次划分后的数据集大小 `m` ，在 `m/2 < k <= m` 的范围内的 `k` ，划分结束得到的子集相同，所以出现了多个 `k` 对应的 `GCP` 相同的情况。

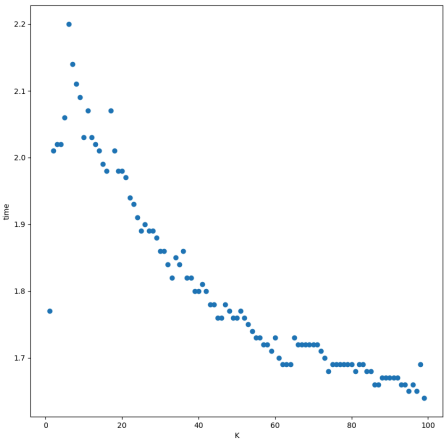
K对RunTime的影响



relax:K对RunTime的影响



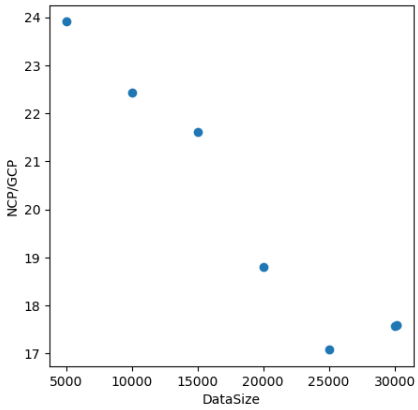
strict:K对RunTime的影响



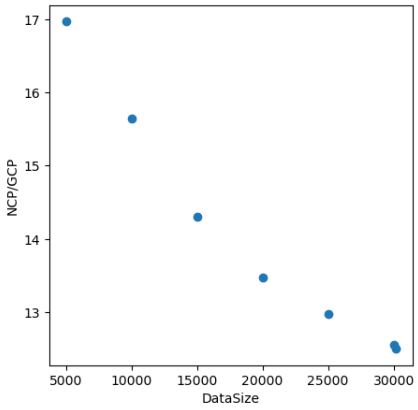
Mondrian2:K对RunTime的影响

- **分析方法**：保持 qi 和 data_size 不变，使 k 的值从 1-100 递增，记录 RunTime ，画出散点图。
- **结果**：随着 k 的增大， RunTime 逐渐变小。
- **分析原因**： Mondrian 算法采用贪心方法，不断对划分得到的数据集再次进行划分，直至不满足 k 匿名。因此 k 越小代表着划分越精细，划分次数越多，故所需要的时间越长。对于 relax 型划分方式，同上，相邻不同的 k 对应的划分结果相同，故步骤次数相同，运行时间几乎相同。

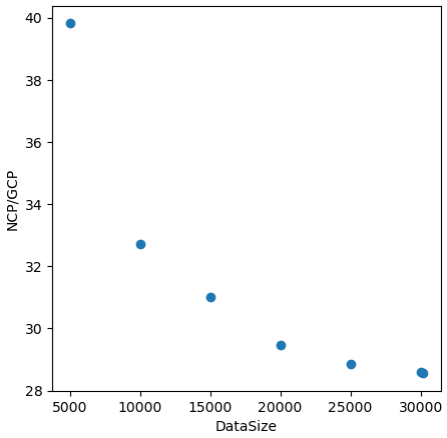
DataSize对GCP的影响



relax:DataSize对GCP的影响



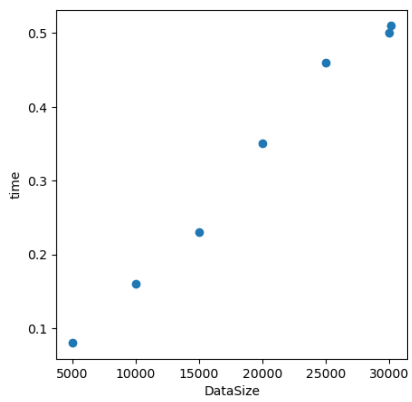
strict:DataSize对GCP的影响



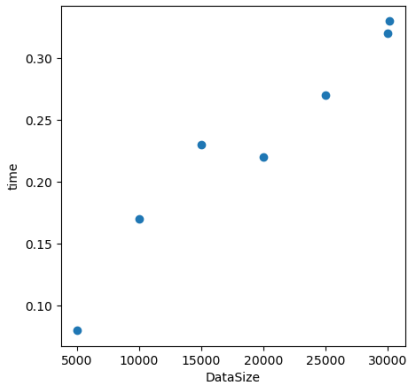
Mondrian2:DataSize对GCP的影响

- **分析方法**：保持 k 和 qi 不变，改变 data_size 的大小，记录 GCP ，画出散点图。
- **结果**：在误差允许的范围内，在本数据集基础上，随着 data_size 的增大， GCP 大小有减小趋势，但若数据分布不均匀， GCP 可能会增大。
- **原因分析**：
  - 对于数值型数据，每条数据t在第i个维度对应的 NCP 为：$NCP_{A_i}t = \frac{z_i - y_i}{|A_i|}$，其中$|A_i|$对应该维度数值的最大最小值之差，$z_i - y_i$为划分后的子集的区间大小。因此，随着 data_size 的增大，对于某一维度的数据，范围可能会变大或者不变。在 k 不变时，由于数据量的增多，故$z_i - y_i$也有可能变小或者不变。变小是因为位于该区间的数据变多，可以再次进行划分。故每个数据在某一维度对应的 NCP 可能会减小，所有数据所有维度的 NCP 平均也会变小。
  - 对于非数值型数据，$NCP_{A_i}t = \frac{|u|}{|A_i|}$ 。由于数据量变大，则可能对于该属性，属性值变多，则$|A_i|$变大或者不变，与数值型数据同理，可泛化为 u 的数据变多，若 u 不是叶子节点，则可能进行进一步划分，从而导致$|u|$变小或者不变，综上，计算得到的 GCP 有变小的趋势。
  - 对于 relax 型划分结果，出现 GCP 相对增加的点原因可能因为数据分布不均匀。

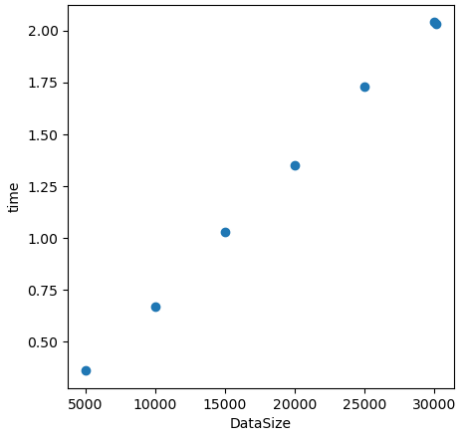
DataSize对RunTime的影响



relax:DataSize对RunTime的影响



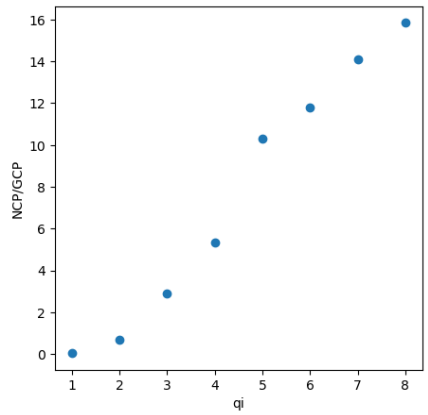
strict:DataSize对RunTime的影响



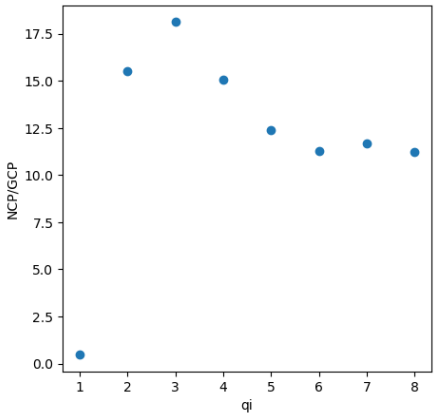
Mondrian2:DataSize对RunTime的影响

- **分析方法**：保持 k 和 qi 不变，改变 data_size 的大小，记录 RunTime ， 画出散点图。
- **结果**：在误差允许的范围内，在本数据集基础上，随着 data_size 的增大， RunTime 大小总体有增大趋势。
- **原因分析**：在其他方面相同的情况下， data_size 越大，意味着要满足 k 匿名可能需要更多的划分次数，则运行时间也就增加。

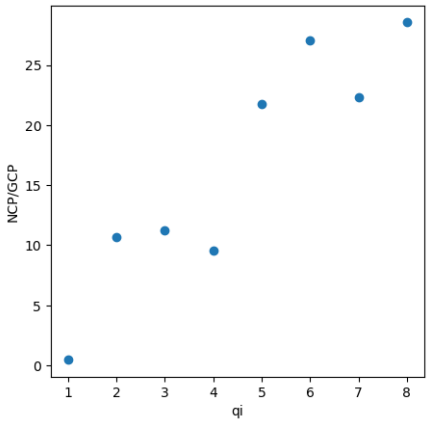
qi对GCP的影响



relax:qi对GCP的影响



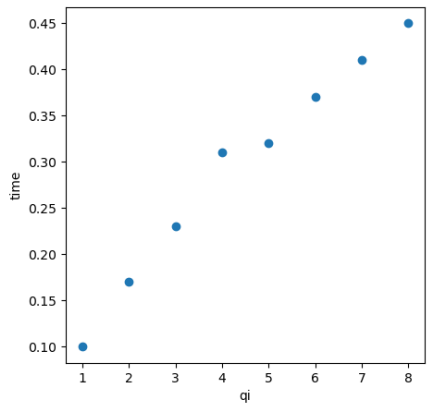
strict:qi对GCP的影响



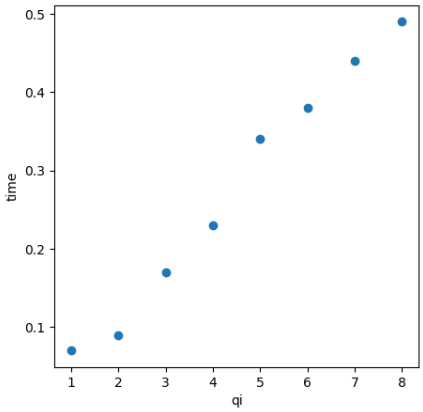
Mondrian2:qi对GCP的影响

- **结果**：综合来看，随着 qi 的增大， GCP 增大
- **分析原因**：在 qi 较小的情况下，划分后的同一子集中，不属于 qi 中的属性值可能不同，就导致信息损失量降低， GCP 降低，故总体来说，随着 qi 的增大， GCP 增大。

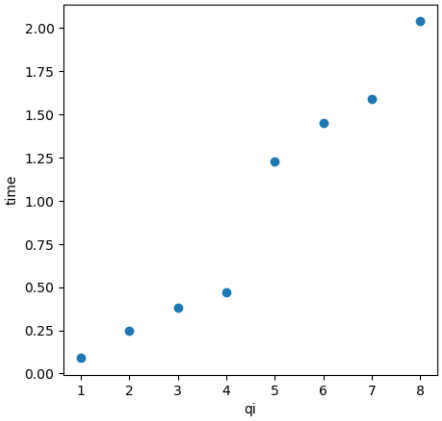
qi对RunTime的影响



relax:qi对RunTime的影响



strict:qi对RunTime的影响



Mondrian2:qi对RunTime的影响

- **结果**：综合来看，随着 `qi` 的增大， `RunTime` 增大
- **分析原因**：在 `qi` 较大时，划分需要考虑的维度更多，划分的计算复杂度增加，故 `RunTime` 随着 `qi` 的增大而增大。

综上所述，在误差允许的范围内， `GCP` 随着 `K` 的增大而增大，随着 `data_size` 的增大而减小，随着 `qi` 的增大而增大。 `RunTime` 随着 `K` 的增大而减小，随着 `data_size` 的增大而增大，随着 `qi` 的增大而增大。

全局损失、执行时间、安全性之间的关系

三者主要与 `k` 有关

1.全局损失和执行时间：在利用Mondrian算法进行K匿名时，当K越大，意味着划分后的结构越粗糙，比如说，数值型划分得到的区间范围较大，非数值型数据泛化后的结果在泛化树中层次更高。在此过程中，数据的划分和泛化带来了信息的损失，而k越大，划分后的数据结果越模糊，信息损失越大。``k`增大，运行时间减小，故全局损失较小需要的运行时间更长。

2.全局损失和安全性：Mondrian算法通过对数据的划分实现k匿名，当k值较大时，意味着每一个划分得到的子集里数据量越大，则根据对应的准标识符及其外部信息，越难找到真正的数据，数据的安全性越高，对应的全局损失也越大。故较大的全局损失会带来较高的安全性。

3.执行时间和安全性：安全性高， `k` 较大，执行时间较短。

综上所述，更高的安全性会带来更大的全局损失和执行时间。全局损失和执行时间可以兼顾。所以，要在安全性和信息损失以及执行时间上找到一个平衡点，可以通过 `Mondrian` 算法合理选择 `k` 等。

Mondrian算法的特点

1. 采用贪心算法，递归划分数据集
2. 划分方式无法保证信息损失最小，不能在保障安全性的同时保障信息损失小和运行效率
3. 可以有效对一维和多维数据进行k匿名，同时可以有效处理非数值型数据，具可扩展性。
4. 通过 `datasize` 和 `qi` 对 `RunTime` 的影响来看，该算法能较快速处理一定规模的多维数据，但对于高维数据处理能力相对较弱。

代码

relax

```
def anonymize_relaxed(partition):
    """
    recursively partition groups until not allowable
    """
    if sum(partition.allow) == 0:
        # can not split
        RESULT.append(partition)
        return

    # choose attribute from domain (*)
    dim = choose_dimension(partition)
    # 若dim = -1， 说明前面在数据的range等计算时有误
    if dim == -1:
        print("Error: dim = -1 (anonymize_relaxed function)")
        # need to debug
        pdb.set_trace()

    # use frequency set to get median (*)
    """
    # use find_median function
    # return value : split_val: 中位数
    # next_val: 中位数的右侧数字
    # low: 该维度的最小值
    # high: t该维度的最大值
    """
    # split_val, next_val可能是''
    # low_val和high_val也可能是''
    (split_val, next_val, low_val, high_val) = find_median(partition, dim)

    # update parent low and high (*)
    # 找到对应low_val和high_val的位置（在值排序后的位置）
    if low_val != '' and high_val != '':
        partition.low[dim] = QI_DICT[dim][low_val]
        partition.high[dim] = QI_DICT[dim][high_val]
    # 参考mondrian_2.py，对split_val进行判断
    if split_val == '':   # 说明该维无法继续分
        partition.allow[dim] = 0
        anonymize_relaxed(partition) # 继续寻找可划分的下一维度
        return

    # split the group from median (records equal to the mean are stored first) (*)
    # 找到了用于划分的val和nextval，但由于是平分，所以对于值和split_val相同的要单独划分
    eq_split_val_list = []
    l_val_list = []
```

```
r_val_list = []
mean = QI_DICT[dim][split_val]
for record in partition.member:
    # 根据QI_DICT的值进行划分，如果该值对应的位置(value)小于split_val对应的位置，则在左侧
    record_index = QI_DICT[dim][record[dim]]
    if record_index < mean:
        l_val_list.append(record)
    elif record_index > mean:
        r_val_list.append(record)
    else:
        eq_split_val_list.append(record)
# handle records in the middle (*)
# 划分eq_split_val_list
lhs_len = len(l_val_list)
# rhs_len = len(r_val_list)
########
#  若要使得数量为奇数时，左划分数量多1，则需要变为 half_len =(len(partition)+1)//2 ，
# 鉴于在求中间数find_median的时候中间数取值为(len(partition))//2
# 所以在此处使用half_len =(len(partition)+1)会有错误。这种方式对应（lhs + 1） | lhs = rhs
#######
half_len = (len(partition)) // 2

for i in range(half_len - lhs_len):
    record = eq_split_val_list.pop()
    l_val_list.append(record)

rhs_low_val = QI_DICT[dim][next_val]
# the rest will be added to rhs
if len(eq_split_val_list) > 0:
    rhs_low_val = QI_DICT[dim][split_val] # 对于split_val有一些被分到了右侧，更新右侧的low
    for record in eq_split_val_list:
        r_val_list.append(record)

# check is lhs and rhs satisfy k-anonymity (*)
# 在splitVal中，已经对是否满足进行了判断，在这里的判断并不影响
if len(l_val_list) < GL_K or len(r_val_list) < GL_K:
    # 不进行下一维度划分，因为数据集总长度小于2k
    partition.allow[dim] = 0
    # print("Error: split doesn't satisfy k-anonymity (anonymize_relaxed function)")
# anonymize sub-partition
# 创建新的partition，对于左侧的partition，该维度的high被改变，右侧的partition，该维度的low被改变
lhs_high = partition.high[:]
rhs_low = partition.low[:]
lhs_high[dim] = mean
rhs_low[dim] = rhs_low_val
lhs = Partition(l_val_list, partition.low, lhs_high)
rhs = Partition(r_val_list, rhs_low, partition.high)
```

```
anonymize_relaxed(lhs)
anonymize_relaxed(rhs)
```

strict

```
def anonymize_strict(partition):
    """
    recursively partition groups until not allowable
    """
    allow_count = sum(partition.allow)
    # only run allow_count times
    if allow_count == 0:
        RESULT.append(partition)
        return
    for index in range(allow_count):
        # choose attribute from domain (*)
        dim = choose_dimension(partition)
        # 若dim = -1， 说明前面在数据的range等计算时有误
        if dim == -1:
            print("Error: dim = -1 (anonymize_strict function)")
            # need to debug
            pdb.set_trace()
        """
        # use find_median function
        # return value : split_val: 中位数
        # next_val: 中位数的右侧数字
        # low: 该维度的最小值
        # high: t该维度的最大值
        """
        (split_val, next_val, low_val, high_val) = find_median(partition, dim)

        # update parent low and high (*)
        if low_val != '' and high_val != '':
            partition.low[dim] = QI_DICT[dim][low_val]
            partition.high[dim] = QI_DICT[dim][high_val]
        # 参考mondrian_2.py，对split_val进行判断
        if split_val == '' or split_val == next_val:   # 说明该维无法继续分
            partition.allow[dim] = 0
            # anonymize_strict(partition)
            continue   # 与anonymize_relaxed不同，该函数用循环写，所以continue

        # split the group from median (*)
        l_val_list = []
        r_val_list = []
        # 找到splitVal在数据集中的排序位置
        mean = QI_DICT[dim][split_val]
        for record in partition.member:
            # record might not be sorted, and the value might not a number, so we can use QI_DICT
            # find this record's index
            # 由于是严格划分，则小于等于mean的都划分到左侧
            record_index = QI_DICT[dim][record[dim]]
```

```
            if record_index <= mean:
                l_val_list.append(record)
            else:
                r_val_list.append(record)
        # check is lhs and rhs satisfy k-anonymity (*)
        if len(l_val_list) < GL_K or len(r_val_list) < GL_K:
            # print(f"Error: split dim{dim+1} doesn't satisfy k-anonymity (anonymize_strict function)")
            # 该维度不可进行划分
            partition.allow[dim] = 0
            continue
        # anonymize sub-partition
        lhs_high = partition.high[:]
        rhs_low = partition.low[:]
        # 更新划分子集对应的low和high
        lhs_high[dim] = mean
        rhs_low[dim] = QI_DICT[dim][next_val]
        lhs = Partition(l_val_list, partition.low, lhs_high)
        rhs = Partition(r_val_list, rhs_low, partition.high)
        anonymize_strict(lhs)
        anonymize_strict(rhs)
        return
    RESULT.append(partition)
```

Mondrian2

```
def split_categorical(partition, dim, pwidth, pmiddle):
    """
    split categorical attribute using generalization hierarchy
    """
    sub_partitions = []
    # categoric attributes
    splitVal = ATT_TREES[dim][partition.middle[dim]]
    sub_node = [t for t in splitVal.child] # 找到子节点
    sub_groups = []
    for i in range(len(sub_node)):
        sub_groups.append([])
    if len(sub_groups) == 0:
        # split is not necessary
        return []
    # split partition and distribute records  (*)
    # partition.member 记录数据
    # splitVal 是该数据集对应的泛化属性在泛化树中的结点
    # sub_node 是splitVal下一层的泛化属性
    for member in partition.member:     # 按子节点分类，泛化层次降低
        attribute = member[dim]
        # 找到数据集中某条数据第dim维的值属于的sub_node
        for i in range(len(sub_node)):
            try:
                # 如果子节点中有该属性，则把该属性加入该子集
                # node.cover是字典类型，所以考虑KeyError，  类似frequency_set
                sub_node[i].cover[attribute]
                sub_groups[i].append(member)
                break
            except KeyError:  # 属性值不能泛化为sub_node[i]，继续找下一个
                continue
        # 说明不属于上述类别中的任何一个
        if i >= len(sub_node):
            print ("Error generalization(split_categorical)!")
    # 检测是否符合k匿名
    for sub_group in sub_groups:
        if len(sub_group) < GL_K and len(sub_group) > 0:
            return sub_partitions
    # update
    for i in range(len(sub_groups)):
        if len(sub_groups[i]) == 0:
            continue
        # 更新第dim维的 width 和 middle，不能直接在原列表上改，会对后续结果有影响
        dim_width, dim_middle = pwidth[:], pmiddle[:]
        dim_width[dim] = len(sub_node[i])
        dim_middle[dim] = sub_node[i].value
        # 与数据型类似，按照一个维度分过之后，再继续细分
```

```
    sub_partitions.append(Partition(sub_groups[i], dim_width, dim_middle))
return sub_partitions
```