Report

Candidate Generation 的算法思路

由Collision Counting LSH方法我们知道，需要实现的`c2lsh`函数将每一个数据的哈希码和query的哈希码之间计算Collision Counting，并同时保证大于等于α_m的数据被加入到Candidate集合．如果在当前Collision Counting的计算阈值下Candidate集合大小可以达到预定的β_n，则当前的Candidate集合就是所求的，请注意我们需要找到一个最小阈值，其计算下Candidate集合大小达到β_n．

就像课件中Virtual Rehashing所描述的，如果 $h(o) = h(q) \pm i$ 仍然不能获得足够的candidates，那么 $i \to i + 1$，并继续搜索，这个过程保证了每次找到的都是当前最小计算阈值．

实现细节(implementation details)

- 首先对于两个哈希码，我使用map映射的函数`count`来计算他们之间的Collision Counting:

  该函数将每个位置在offset范围内的个数计算出后返回`(id, counter)`，完全依照课件里算法的说明．

```
def count(hashes_1, hashes_2, offset):
    counter = 0
    for hash1, hash2 in zip(hashes_1[1], hashes_2):
        if myabs(hash2 - hash1) <= offset:
            counter += 1

    return (hashes_1[0], counter)
```

- 接下来对于每一个数据的哈希码，我都将其和query的哈希码进行Collision Counting计算，这里我采用了Spark机制下的map函数对RDD数据进行操作:

```
tmp_data_hashes = data_hashes.map(lambda x: count(x, query_hashes,
offset))
```

  然后对于 $\geq \alpha_m$ 的数据我进行了过滤，并提取该数据的ID:

```
tmp_data_hashes = tmp_data_hashes.filter(lambda x: x[1] >=
alpha_m).map(lambda x: x[0])
```

  最后计算candidate集合的大小是否达到β_n，如果没有则返回`None`

- 在`c2lsh`主函数中我采用了更优的offset搜索策略，使其在 $\mathcal{O}(log(n))$ 时间复杂度下能找到最优的offset，其中n是最大的可行offset．这将在下一节优化方案中提到．

```
def c2lsh(data_hashes, query_hashes, alpha_m, beta_n):
    l, r = 0, 2
```

```
    # I've used binary search, and here the upper bound will be
determined:
    while sol(data_hashes, query_hashes, r, alpha_m, beta_n) == None:
        l = r
        r *= 2
    res = r
    # binary search with upper bound `r` and lower bound `l`:
    while l <= r:
        mid = l + ((r - l) >> 1)
        tmp = sol(data_hashes, query_hashes, mid, alpha_m, beta_n)
        if tmp == None:
            l = mid + 1
        else:
            res = mid
            r = mid - 1

    return sol(data_hashes, query_hashes, res, alpha_m, beta_n)
```

优化方案

注意到搜索最优offset的过程，即在一个无上界的区间上搜索最小的满足条件的值.

我使用二分搜索，首先确定该问题offset的上界，我采用指数的方式搜索解空间，即找到一个i，使$2^i < offset < 2^{i+1}$，并且$offset = 2^{i+1}$是可行的，$offset = 2^i$是不可行的.

这样解空间就缩小到$2^i \sim 2^{i+1}$上，对该区间使用二分搜索即可.

该优化方案对offset较大时效果明显，在 $\mathcal{O}(log(n))$ 时间复杂度下能找到最优的offset，其中n是一个可行offset的值.

读取自己的测试用例并测试

我的测试用例基于`toy/toy_hashed_data`，即复制一次`toy/toy_hashed_data`，观察输出结果的变化来证明程序正确性.

首先老师提供的测试用例`toy/toy_hashed_data`中，输出结果应为:

```
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Number of candidate:  10
set of candidate:  {0, 70, 40, 10, 80, 50, 20, 90, 60, 30}
```

因为ID为0, 10, 20, 30 ... 90的数据分别是:

```
(0, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 40, 40, 40, 40, 40, 40 ...])
(10, [1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 40, 40, 40, 40, 40, 40 ...])
(20, [1, 1, 1, 1, 1, 1, 1, 1, 2, 3, 40, 40, 40, 40, 40, 40 ...])
...
```

经过验算发现这十个确实是符合要求的.

当我们把如上`toy/toy_hashed_data`复制一遍后，可以发现ID为0的数据出现了两次，且为0和100，同理ID为10的数据被复制为ID等于10和110. 由此最后得到的十个结果应为: 0, 100; 10, 110; 20, 120; 30, 130; 40, 140;

```
Number of candidate:  10
set of candidate:  {0, 130, 100, 40, 10, 140, 110, 20, 120, 30}
```

但是此时的offset相较与老师的测试用例会变小，正好满足ID为40的数据 与 query数据进行count.

运行环境与指令

- 操作系统: Ubuntu 18.04.3 LTS
- Spark version 3.0.0
- Python 3.6.5

`run.py` 的内容就是 `COMP9313 Project 1-specs.ipynb`里面的内容.

```
$ python3 run.py
```

运行自己的测试用例请将`run.py`里的代码改为:

```
with open("my_toy_hashed_data", "rb") as file:
    data = pickle.load(file)
```